Javaで学ぶ 数値解析

和光システム研究所 著

東京電機大学出版局

Javaで学ぶ 数値解析

和光システム研究所 著

東京電機大学出版局

はじめに

本書の主な目的は二つあり，第一は数値解析を Java で初めて学びたいという初学者に対する数値解析の解説であり，第二はすでに Fortran その他の言語で数値解析を理解しているが Java はとりつきにくいと感じている，数値解析の熟達者に対する Java への招待である．そこで，本書の読者層はいろいろ想定され，理工系の学部学生から，計算機を扱える研究者にまで，幅広くわたるであろう．著者は 1960 年代初頭より，数値計算を扱ってきたので，手続き型言語の Fortran 等による膨大なプログラムの蓄積を保存しており，このたび，それを Java 言語に翻訳して解説する運びとなった．Java はオブジェクト指向言語であるから，もちろん Fortran からの一対一対応のある翻訳は不可能であるが，必要以上にオブジェクト指向言語固有のトリッキーな書き方は抑えて，Java に書き直してある．実際の数値計算で大切なことは，効率 (計算の速さ) と精度である．しかし，この二つを強調しすぎると，解法のアルゴリズムが理解しにくくなる面もあるので，本書では，アルゴリズムを中心に述べ，必要に応じて，効率と精度について詳述している．

本書の特徴は以下の 2 点である．第 1 点は，1980 年代にグラフィックスの国際規格として制定された GKS の図形出力などの機能を Java のグラフィックス機能を用いてクラス化し，それといくつかの追加機能を使って多くのプログラムの出力をグラフィックス化したことである．このクラスパッケージ mygks は，まだ改善の余地もあるので，β 版として無償で提供することになった．特徴の第 2 点は，手続き型言語の副プログラム単位に相当する部分は，同じように独立なクラスの単位に収め，それを使うテストプログラムから独立させて，パッケージ化してあることである．さらに，目的に応じてテストプログラムの一つを少し変更するだけで期待するプログラムが書けるように心がけてある．また，ほとんどすべてのテストプログラムは，mygks のクラスを引用して図を表示するようになっているが，数値解析部分と表示部分はできるだけ切り分けて書かれているから，表示部分が不要の場合には簡単に削除することができる．

このように，本書で描かれている図は，Java の GKS タイプの図形出力クラスを使って描かれており，説明も Java プログラムを使って書かれているが，数値解析の解説それ自体は，特に言語にしばられることはないので，他の言語でも GKS パッケージが使えれば，参考になるであろう．

最後に，いろいろ重要な助言をいただいた，旧 図書館情報大学 (現 筑波大学図書館

情報専門学群）松本紳教授，時井真紀助手，卒業研究でともに学んだ多くの卒業生諸君に感謝いたします．また，東京電機大学出版局の松崎真理氏をはじめ，出版局の皆様に感謝いたします．

2004 年聖夜

三軒茶屋にて　　著者

目次

第 1 章　数値解析と誤差　1

1.1　打ち切り誤差 1
1.2　刻み幅誤差 2
1.3　桁落ち誤差 4
1.4　漸化式で生じる桁落ち誤差 (球ベッセル関数) 5
1.5　丸め誤差 7
1.6　計算の順序によって生じる誤差 8
演習問題 9

第 2 章　ソート，マージ　10

2.1　シャッフル 10
2.2　最大値 12
2.3　バブルソート 13
2.4　櫛ソート 15
2.5　選択ソート 16
2.6　挿入ソート 17
2.7　シェルソート 18
2.8　クイックソート 20
2.9　ヒープソート 22
2.10　マージ 24
2.11　マージソート 26
演習問題 28

第 3 章　補間法　29

3.1　ラグランジュの補間法 29
3.2　ラグランジュ補間の汎用プログラム 34
3.3　ニュートンの補間法 37
3.4　エルミートの補間法 39
3.5　スプライン補間 43
3.5.1　B スプラインによる補間 43

3.5.2　3 次スプライン補間 52
演習問題 57

第 4 章　方程式の解法　58
4.1　代数方程式 58
4.1.1　2 次方程式 58
4.1.2　3 次方程式 (カルダノ公式) 61
4.2　2 分法 63
4.3　挟み撃ち法 64
4.4　反復法 67
4.4.1　ニュートン・ラフソン法 67
4.4.2　反復法の一般論 70
4.5　逆 2 次関数法 70
4.6　多数の解がある場合の汎用プログラム 72
演習問題 74

第 5 章　数値積分法　75
5.1　矩形近似 75
5.2　ニュートン・コーツ型積分 76
5.2.1　台形近似 76
5.2.2　シンプソンの公式 77
5.2.3　一般型 79
5.3　ガウス型積分 81
5.3.1　区間が $[-1, 1]$ で重み関数が $w(x) = 1$ の場合 82
5.3.2　区間が $[-1, 1]$ で重み関数が $w(x) = \frac{1}{\sqrt{1-x^2}}$ の場合 86
5.3.3　区間が $[0, \infty)$ の場合 88
5.3.4　区間が $(-\infty, \infty)$ の場合 90
5.4　誤差の評価 93
演習問題 94

第 6 章　常微分方程式の解法　95
6.1　1 階常微分方程式 95
6.2　一段階法 97
6.2.1　オイラーの前進公式 97
6.2.2　ルンゲ・クッタ法 2 次公式 98
6.2.3　ルンゲ・クッタ法 4 次公式 100
6.3　多段階法と予測子・修正子法 100
6.4　高階常微分方程式 (人工衛星の軌道計算) 101

6.4.1 連立 1 階常微分方程式 102
6.4.2 人工衛星，惑星の軌道計算 (ルンゲ・クッタ法 4 次公式) 103
6.5 ルンゲ・クッタ・ジル法 (水素原子の電子状態) 105
演習問題 110

第 7 章 連立 1 次方程式の解法 111

7.1 連立 1 次方程式 111
7.1.1 ガウスの消去法 112
7.1.2 掃き出し法 115
7.1.3 LU 分解 116
7.2 行列式 119
7.2.1 クラマーの公式 119
7.2.2 行列式の値 120
7.3 逆行列 123
演習問題 124

第 8 章 関数近似と平滑化 125

8.1 最小二乗法 125
8.1.1 1 個の変数に対して線形関係がある場合 126
8.1.2 変数が 2 個以上ある場合 128
8.1.3 変数 $\boldsymbol{x}$ の 2 次式の場合 129
8.2 チェビシェフ多項式近似 130
8.3 データの平滑化 135
8.3.1 サビツキー・ゴーレイ法 135
8.3.2 B スプラインによる平滑化 138
8.3.3 フーリエ変換を使う平滑化 142
演習問題 143

第 9 章 固有値問題 144

9.1 主軸問題 144
9.2 固有値と固有ベクトル 146
9.2.1 固有値問題 147
9.2.2 直接法 148
9.2.3 ヤコビ法 149
9.2.4 累乗法 153
9.2.5 QR 法 157
演習問題 165

第 10 章　フーリエ級数とフーリエ変換　166

10.1　フーリエ級数 166
10.1.1　三角関数による級数展開 168
10.1.2　指数関数 e^{ix} による展開 169
10.2　フーリエ変換 170
10.3　高速フーリエ変換（FFT） 172
10.4　3 次元極座標系におけるフーリエ変換 176
10.5　畳み込みと畳み込み除去 180
10.5.1　畳み込み 181
10.5.2　畳み込み除去 185
演習問題 187

第 11 章　乱数　188

11.1　モンテカルロ法と積分 188
11.2　関数型分布乱数 190
11.2.1　標準指数分布乱数 190
11.2.2　ローレンツ型分布乱数 192
11.2.3　正規分布乱数 193
11.2.4　球面一様分布乱数 195
11.2.5　表読み取り法 196
11.3　シミュレーション 200
11.3.1　ビュフォンの針 200
11.3.2　仮想浅間降灰予測 202
演習問題 204

第 12 章　スプライン関数と作図　205

12.1　切断べき関数 205
12.2　B スプライン 207
12.3　リーゼンフェルトの作図 208
12.3.1　2 次元開曲線 209
12.3.2　重複データ点 213
12.3.3　2 次元閉曲線 214
演習問題 216

付録 A　Fortran ユーザを Java にご招待　217

A.1　Fortran と Java の実行 218
A.1.1　Fortran プログラムの実行 218
A.1.2　Java プログラムの実行 218

A.2 入出力 219
A.2.1 出力 219
A.2.2 入力 220
A.2.3 ファイルからの入力 221
A.3 Fortran から見た Java の主な相違点 223
A.4 Math クラスの主なメソッド 228
A.5 アプリケーションとアプレット 229
A.6 ライブラリパッケージの作り方と使い方 233
A.7 インタフェースの使い方 234

付録 B mygks メモ 236

付録 C ソース＆クラスリスト 240

あとがき 247

参考文献 249

索引 250

本書に掲載したソースファイルとパッケージ mygks は，ウェブページからダウンロードできます．

東京電機大学出版局ウェブページ http://www.tdupress.jp/

[メインメニュー] → [ダウンロード] → [Java で学ぶ数値解析]

第 1 章

数値解析と誤差

数値解析のプログラムを作るとき，最後に残る問題は誤差の問題である[1]．プログラムが意図したとおりに正しく書かれていなければならないことはいうまでもないが，正しく書かれていても場合によっては誤った答えを与えることがある．誤差になる原因を大まかに分類すると，最初の入力データに隠れている誤差，近似式の打ち切り誤差，刻み幅誤差，桁落ち誤差および丸め誤差である．計算の途中で誤差が生じていれば，その後の計算にとっては，その誤差が入力データの誤差になり，その伝播が問題となる．

入力データや，近似パラメータなどの $(x_1, \cdots, x_n)$ に隠れている誤差 $(\Delta x_1, \cdots, \Delta x_n)$ があれば，最後の答え $y = f(x_1, \cdots, x_n)$ にも誤差 Δy ができる．それはだいたい

$$\Delta y = \frac{\partial f}{\partial x_1}\Delta x_1 + \frac{\partial f}{\partial x_2}\Delta x_2 + \cdots + \frac{\partial f}{\partial x_n}\Delta x_n \tag{1.1}$$

で与えられる．この式を具体的に評価するためには，関数 $y = f(x_1, \cdots, x_n)$ が必要であるが，それがわからないことが多いので，一般的には x_i の値を少し変えて計算しなおし，y の変化を調べることになる．結局，$\dfrac{\partial f}{\partial x_i}$ の大きな x_i については，特に気を付けねばならないということである．

他方，データやパラメータが正確であればこの誤差は無視できるであろうから，数値解析で主に問題とすべき誤差は残りの四つである．

1.1 打ち切り誤差

電卓も含めて，最近の計算機には四則演算や開平算などの基本的な演算はハードウェアとして組み込まれている．しかし，複雑な関数の場合には，正確な式で計算できることもあるが，それを級数展開して有限個の項のみを使って近似することが多い．今では，三角関数なども，ほとんどのプログラミング言語のソフトウェア，あるいは

[1] 初学者であったり，誤差のことを気にかけなくて済むなら，この章は読み飛ばしてもよいであろう．

ハードウェアで提供されているから，それを使えば済むので，ユーザが自分でプログラムを書くことはほとんどないであろうが，打ち切り誤差の説明として，原点の周りの $\cos(x)$ について調べることにする．$\cos(x)$ を級数展開すると

$$\cos(x) = 1 - \frac{1}{2!}x^2 + \frac{1}{4!}x^4 - \frac{1}{6!}x^6 + \frac{1}{8!}x^8 - \cdots \tag{1.2}$$

である．$|x| < 0.1$ の範囲では，右辺の各項は順次 x^2 で小さくなるから，項を 1 項多くとるたびに，x^2 の値だけでも精度が 2 桁以上良くなり，係数自体も急激に小さくなっていく．近似式を使う x の範囲に応じて収束の速さが異なるから，要求する精度と x の範囲を考慮して使う項数を決めなければならない．$|x| > 1$ になると，項が進むに従って係数は小さくなっていくが，x^2 の値は大きくなっていくから収束が悪く，級数展開を使うのは得策でないといえる．図 1.1 は，$\cos x$ を式 (1.2) の第 1 項のみの近似から，第 2, 3, 4, 5 項までとった近似式によるグラフを描いたものである．原点付近では，項数を増やすにつれてしだいに良くなっていくことが見てとれるであろう．
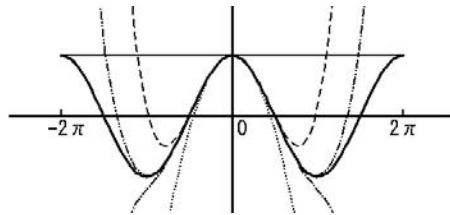
図 1.1　$\cos(x)$ の級数展開．太い実線は $\cos(x)$，細い実線は第 1 項のみ，点線は第 2 項まで，破線は第 3 項まで，一点鎖線は第 4 項まで，二点鎖線は第 5 項までとったときの近似曲線である．

1.2　刻み幅誤差

刻み幅誤差は数値積分や微分方程式の解法などで，x 軸に沿って処理を進めていくとき，1 回の処理を行う x 軸上の幅を粗くしすぎると顕著になる誤差である．この誤差は打ち切り誤差と相補的な関係にあり，積分や微分方程式の計算公式に出てくる級数の項数を，多くとる良い近似では幅を粗くしてもよく，項数を少ししかとらない粗い近似では幅を細かくとらなければならない．台形公式とシンプソンの公式による積分[2]を例として，刻み幅と精度の関係を調べてみる．台形公式は，1 刻みでは被積分関数を 1 次式で近似するものであり，シンプソンの公式は 2 次式で近似するものである．図 1.2 は関数 $f(x) = 8\sqrt{x(1-x)}$ を，x の範囲 $[0, 1]$ で積分するときの刻みの数を横

2. 詳しくは第 5 章を参照のこと．
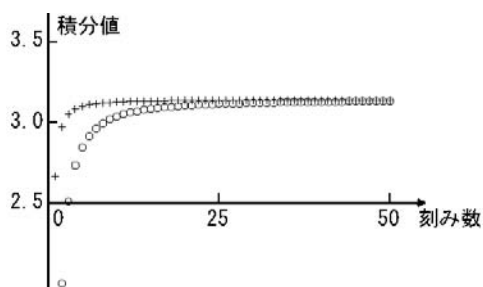
図 1.2　台形公式とシンプソンの公式による関数 $f(x) = 8\sqrt{x(1-x)}$ の範囲 $[0,1]$ での積分．横軸は刻みの数，縦軸は積分値である．台形公式によるものは ∘，シンプソンの公式によるものは + である．

軸に，答えの積分値を縦軸にプロットしたものである．台形公式によるものが ∘，シンプソンの公式によるものが + で示されている．正しい積分値は $\pi = 3.14159265\cdots$ である．

図 1.2 を見ると，シンプソンの公式による場合には刻み数が 20 程度で，ほぼ π に近づいているが，数値的にはまだ満足できるものではない．表 1.1 に刻み数と，両公式による積分値が示されている．シンプソンの公式の場合には，初めの 4, 5 回で 3.1 になるが，台形公式の場合には，20 回近くでやっと 3.1 になり，回数が増えても両者の精度には 2, 3 桁の差がある．両者とも刻み数が 1 桁増えると精度が 1 桁上がる程度で，収束はあまり速くない．

表 1.1　台形公式とシンプソンの公式による積分値と刻み数．関数は $f(x) = 8\sqrt{x(1-x)}$ で，積分範囲は $[0,1]$ である．最も下の行には答え π の真値が載せてある．

刻み数	台形	シンプソン
1	0.0	2.6666
2	2.0	2.9760
3	2.5141	3.0520
5	2.8476	3.1000
10	3.0370	3.1720
25	3.1150	3.1379
50	3.1321	3.1402
100	3.1382	3.1411
1000	3.14148	3.14157
10000	3.141589	3.1415921
100000	3.1415925	3.14159263
π		$3.14159265\cdots$

1.3　桁落ち誤差

プログラム上のアルゴリズムからは気が付かないが，実際に計算してみると大きな誤差が入ってくることがある．その原因の一つは，桁落ち誤差である．計算機でも数値はある限られた桁数で扱われるので，ほとんど等しい二つの数の差をとると有効桁数が減少するために，精度が悪くなってしまう．極端な例として，理論上は正確に 1 であるべき 2 個の数があり，10 進 7 桁程度を扱う計算機で，たまたま $a = 1.000001$ と $b = 0.999999$ であったとする．その差は理論上はゼロでなければならないが，計算機では $a - b = 0.000002 = 2 \times 10^{-6}$ になってしまう．元々それぞれ 6 桁の精度があったのに引き算の結果は誤差だけになり，この結果を使って計算を続けていくことは非常に危険である．図 1.3 は，ほとんど等しい二つの項 $\sqrt{1+x^2}$ と 1 の差が，関数

$$y(x) = \sqrt{\sqrt{1+x^2} - 1} \tag{1.3}$$

の計算で生じる相対誤差

$$\Delta y = \frac{\text{単精度計算の } y(x) - \text{倍精度計算の } y(x)}{\text{倍精度計算の } y(x)} \tag{1.4}$$

を示した図である．最近の計算機では単精度は 10 進約 7 桁，倍精度は約 16 桁の精度があるから，ここでは倍精度計算を真値として使うことにする．

図 1.3 からわかるように，$0.001 < x < 0.002$ ではほとんど 0 になってはいるが，0.001 に近づくにつれ振動が始まっており，精度は 2〜3 桁しかないであろう．x が 0.001 程度以下になると精度は急速に悪くなり，$x = 0.00025$ あたりから下では -1 になっている．このグラフは誤差のグラフであるから，ハードウェアのアーキテクチャや平方のアルゴリズムなどに依存しており，同じプログラムを使っても，まったく異なる結果になることもあろうが，ゼロに近づくに従って悪くなるという一般的な傾向は同じである．式 (1.3) の場合には，分子の有理化をした式

$$y = \sqrt{\frac{x^2}{\sqrt{1+x^2}+1}} \tag{1.5}$$
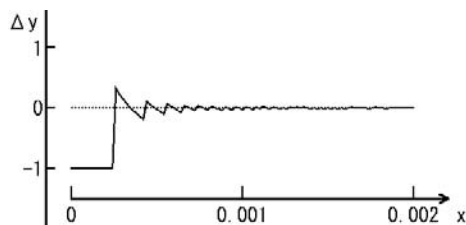
図 1.3　桁落ち誤差の例．実線は桁落ちの起こる式 (1.3) を使ったときの相対誤差の式 (1.4) であり，点線は桁落ちの起こらない式 (1.5) を使ったときの相対誤差の式である．

を使えば桁落ちを避けることができる．この分子の有理化は，2 次方程式の根の計算などで使われることがある．

1.4　漸化式で生じる桁落ち誤差（球ベッセル関数）

特殊関数をはじめ，多くの関数系には漸化式が存在する．それは，その関数系の最初のいくつかの低次の関数がわかっていれば，より高次の関数を低次の関数から漸次求めることができる式である．多くの関数系の場合に低次からの前進漸化式を使うことができるが，円筒座標系や極座標系を使うとき出てくるベッセル関数系では桁落ちが激しく生じるので，最初に高次の関数を近似的に決めてから，高次から低次への後退漸化式を使って，低次の関数を求めるようにしなければならない．ここでは，球ベッセル関数について述べる．球ベッセル関数 $j_l(z)$ は，次の微分方程式

$$-\frac{1}{z^2}\frac{\mathrm{d}}{\mathrm{d}z}\left(z^2\frac{\mathrm{d}j_l(z)}{\mathrm{d}z}\right)+\frac{l(l+1)}{z^2}j_l(z)=j_l(z) \tag{1.6}$$

の解である．これは，極座標系の動径座標に関する微分方程式であり，変数 z の定義域は $[0,\infty)$ である．次数 l は $l \geq 0$ であり，$l=0$〜4 の関数形は

$$\begin{aligned}
j_0(z) &= z^{-1}\sin z\\
j_1(z) &= z^{-2}(\sin z - z\cos z)\\
j_2(z) &= z^{-3}[(3-z^2)\sin z - 3z\cos z]\\
j_3(z) &= z^{-4}[(15-6z^2)\sin z - z(15-z^2)\cos z]\\
j_4(z) &= z^{-5}[(105-45z^2+z^4)\sin z - z(105-10z^2)\cos z]
\end{aligned} \tag{1.7}$$

である．図 1.4 は式 (1.7) を図示したものである．

これらの関数は，$z\to 0$ の極限では

$$j_l(z)\sim\frac{z^l}{(2l+1)!!} \tag{1.8}$$

であり，$z\to\infty$ での漸近形は

$$j_l(z)\sim\frac{1}{z}\sin\left(z-l\frac{\pi}{2}\right) \tag{1.9}$$
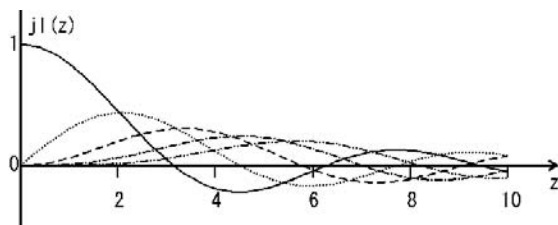
図 1.4　球ベッセル関数 $j_l(z)$．実線は $l=0$，点線は $l=1$，破線は $l=2$，一点鎖線は $l=3$，二点鎖線は $l=4$ である．

である．球ベッセル関数の前進漸化式は

$$j_l^{(f)}(z) = (2l-1)\frac{j_{l-1}(z)}{z} - j_{l-2}(z) \tag{1.10}$$

であるが，式 (1.7) あるいは図 1.4 からわかるように，同じ値の z について，l が大きくなると $j_l(z)$ の値が小さくなるので，この漸化式を使って小さい l から大きい l へと $j_l(z)$ を求めていくと，桁落ち誤差がたちまち大きくなってしまう．そこで，この漸化式 (1.10) を逆に大きい l から小さい l へ降りてくる後退漸化式

$$j_l^{(b)}(z) = (2l+3)\frac{j_{l+1}(z)}{z} - j_{l+2}(z) \tag{1.11}$$

に書きなおす．この式では先のような桁落ちは発生しない．$j_l^{(f)}(z)$，$j_l^{(b)}(z)$ の肩の (f) と (b) は前進 (forward) と後退 (backward) を示すための記号であり，他に特別な意味はない．図 1.5 は式 (1.11) で求めた関数値を真値として式 (1.10) を割り，その絶対値の対数をとった式

$$\Delta E(z) = \log\left\{\left|\frac{j_l^{(f)}(z)}{j_l^{(b)}(z)}\right|\right\} \tag{1.12}$$

である．誤差が発生し始めると，伝播過程でたちまち発散してしまうために，グラフがこの図以上に見にくくなってしまうので，対数をとって発散の値を少しでも緩やかになるようにしているのである．誤差がなければこの値はゼロである．この計算では他の誤差が顕著には入らぬように，すべてを倍精度で行っている．

後退漸化式で使う最初の値とするために，大きな l の $j_{l+2}(z)$ と $j_{l+1}(z)$ の値を正確に計算することは困難である．しかし，各 z に対して，ある l $(=M)$ より大きな l の $j_l(z)$ は無視できるほど小さくなってしまうので，$j_{M+2}(z)$ をゼロとし，$j_{M+1}(z)$ を非
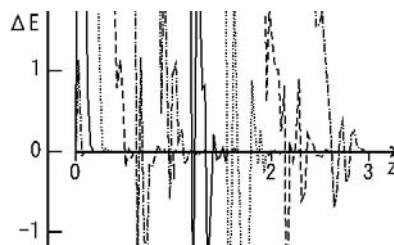
図 1.5　球ベッセル関数の前進漸化式で生じる誤差．誤差がなければこれらの値はゼロである．z がゼロの近傍にある二点鎖線の細長く尖った三角は次数 $l=4$ のグラフであり，z が小さい範囲では $l=4$ ですでにグラフで見れるほどの誤差が生じている．しかし，$z>0.1$ では小さくなっている．同じくゼロ近傍の正の実線は $l=5$ であり，誤差は非常に大きくなるが，これも $z>0.2$ では小さくなっている．ゼロ近傍では $l>5$ のグラフはもっと激しく発散しているが，図からはみ出しており現れていない．z が大きくなるにつれ，小さな l から順に発散が収まり始め，順次大きな l のグラフが図の範囲に見られるようになってくる．$l=5$ の実線の右の点線は $l=6$ であり，次の破線は $l=7$，一点鎖線は $l=8$ というように最も右の一点鎖線の $l=13$ のグラフが現れるところまでが示されている．

常に小さな数 (プログラム上では 10^{-45}) として，式 (1.11) を使えば $l \leq M$ の l について $j_l(z)$ の近似値に比例した値を順次求めることができる．式 (1.11) を最初に適用して求めた $j_M(z)$ はもちろん正しくないが，式 (1.11) を繰り返し用いていくにつれて精度が増してくる．そこで，いま必要とする l の最大の値を L とすると，z と L に応じてとるべき M の値が定まってくる．このようにして求めた $j_l(z)$ に比例した数値は，相対的な値であるから，正しい $j_l(z)$ の値に規格化しなおさねばならない．そのためには正しい $j_0(z)$ を別に求め，それを使って規格化しなおせばよい．$j_0(z)$ を求める式として，$z \leq 0.02$ のときには級数展開 $1 - \frac{1}{6}z^2 + \frac{1}{120}z^4$ を用い，$z > 0.02$ のときには $\frac{\sin z}{z}$ を用いる．$z > 0.02$ で $|\sin z| < 0.1$ となってしまうような z の範囲では $\frac{\sin z}{z}$ の精度に問題が生じるので，$j_0(z)$ の代わりに $j_1(z) = \frac{\sin z}{z^2} - \frac{\cos z}{z}$ を用いて規格化しなおせばよい．

1.5　丸め誤差

計算機で扱う桁数が限られているため，最後の桁にはとかく誤差が入り込んでいる．手計算などでも，円周率の π の値を 5 桁まで使うときには真値 $3.141592\cdots$ の 6 桁目の 9 を四捨五入して (丸めて) 3.1416 とするので，最後の桁は正確であるとはいえなくなっている．計算機の内部では，多くの場合，数値を 2 進数で表現しており，最後の桁 (ビット) はやはり怪しくなっている．10 進数の 0.1 は 2 進数では循環小数であるから，計算機で扱うその値は近似値である．実際，0.1 を繰り返し足し込んでいくと，10,000 回程度でも精度は 3 桁ほど落ちてしまう．図 1.6 は単精度の 0.1 を 10,000 回まで足していく過程で，相対誤差

$$\Delta y = \frac{\displaystyle\sum_{i=1}^{n} 0.1 - 0.1n}{0.1n} \tag{1.13}$$

がどのように増えていくかを示すものである．
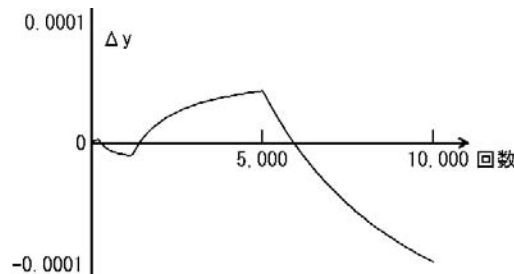
図 1.6　足し算の繰り返しによる丸め誤差の蓄積．横軸は単精度の 0.1 の足し算を繰り返した回数，縦軸は式 (1.13) による丸め誤差の相対値である．

この式の分子の第 1 項は 0.1 を n 回足し算するものであるから，足す過程で丸めの誤差が蓄積されるが，分母などの $0.1n$ は 0.1 を n 倍する計算であるから誤差は増えないので，真値と見なすことができる．繰り返し足していく過程で，桁上がりが起こると最後のビットの状況が変わるので，変化は単調ではないこともわかる．倍精度の計算を行うと，10,000 回程度で精度はやはり 3 桁ほど落ちるが，それでも 12 桁程度は精度があるので，10,000 回程度の足し算では，通常，あまり問題は生じない．

1.6　計算の順序によって生じる誤差

計算機で扱う桁数が限られているために，気を付けなければならないことがもう一つある．これも単精度計算ではたまに起こることである．7 桁ほど大きさの異なる数値の演算は，演算の順序に気を付けないと，小さいほうの数値の効果に悪い影響が出てしまう．例えば，単精度の 1.0 に単精度の 1.0×10^{-7} あるいは 1.0×10^{-8} を 1 万回加えることにする．文章のとおりに，1.0 に 1.0×10^{-7} を加える操作を 1 万回繰り返すと，

$$(\cdots(1.0 + 1.0 \times 10^{-7}) + 1.0 \times 10^{-7})\cdots) + 1.0 \times 10^{-7} \tag{1.14}$$

となり，1.0×10^{-7} を 1 万回加えてから 1.0 に加えると，

$$1.0 + \sum_{i=1}^{10000} 1.0 \times 10^{-7} \tag{1.15}$$

となる．これらの結果はそれぞれ 1.0011921 と 1.001 であり，前者の場合には誤差が 0.0001921 だけ入り込んでいる．このように，単精度の数値は精度が 7 桁程度しかないから，$1.0 + 1.0 \times 10^{-7} = 1.000000119\cdots$ となり誤差が蓄積されていく．また，次の場合は $1.0 + 1.0 \times 10^{-8} = 1.0000000\cdots$ となり 1.0×10^{-8} の寄与がなくなってしまう．そこで，小さい数同士をまず足してから，大きい数と足し合わせなければならないことがわかる．

21 世紀になる頃から計算機の性能は飛躍的に向上し，計算速度が速くなったばかりではなく，実メモリや補助メモリが安価になり大量のデータが使えるようになった．そこで，数値解析を主な目的とする Fortran では倍精度の計算が普通となり，Java では double が標準となっているので，丸めの誤差や桁落ちの誤差なども，以前のようには気にしなくてよくなっている．しかし，打ち切り誤差や刻み幅誤差には，相変わらず注意が必要である．これらは，数式の性質から理論的に決められることもあるが，やはり，パラメータ値を振ってみて，試行錯誤で式 (1.1) を調べることも必要であろう．

演習問題

1.1　図 1.1 のプログラム Z01_01_cos.java を拡張して，式 (1.2) の 7 項までの図を描きなさい．7 項までの式を知らなかったら，自分で調べること．また，プログラム名は Q01_01_Z01_01_cos.java とすること．

解答例： ソースプログラム Q01_01_Z01_01_cos.java

第 2 章

ソート，マージ

一組のデータを，ある順番の規則に従って並べ替えることは，データの大小を比較するだけで計算をするわけではないが，計算機にとって得意とするところである．このことはソート (sort) といわれる．ある順番の規則に従って並べられている複数の組のデータを，その順番の規則に従って並べながら混ぜ合わせることは，マージ (merge) といわれる．ソートとは逆に，不規則に並べたい場合には，乱数を使って並べ替えればよく，ここではシャッフル (shuffle) ということにする．

並べ替えには小さい値のほうが前に来る昇順と，大きい値のほうが前に来る降順があるが，ここではすべて昇順に並べる方法で説明をする．

ソートにはアルゴリズムが簡単な方法として，バブルソート，選択ソート，挿入ソートなどがある．これらの方法では，並べ替えが済んだ後，同じ値をもつ複数のデータの順番が，並べ替えを行う前と入れ替わらないようにすることができる．そこで，このことを安定なアルゴリズムであるという．しかし，所要時間はデータの数 n の 2 乗 n^2 に比例 ($O(n^2)$) し，高速ではない．一方，高速なアルゴリズムとして櫛ソート，シェルソート，クイックソート，ヒープソート，マージソートなどがあり，これらは安定なアルゴリズムではないが，所要時間は $n \log n$ に比例 ($O(n \log n)$) し，高速である．安定な方法では，値を隣同士で 1 個ずつ順次比較して並べ替えていくが，これらの方法では，データ全体を扱いながら，より小さい値を前方に，より大きい値を後方に移していき，最終的に昇順に整列させるので，同じ値のデータの順番が入れ替わることもあり，不安定である．そのほかに，分布数えソート (counting sort)，逆写像ソート (inverse mapping sort)，ラディックスソート (radix sort) など，制限はあるけれども高速な方法もあるが，扱わないことにする．

ソートをするためのデータを作るには，シャッフルが必要であり，まず，シャッフルから始める．

2.1 シャッフル

多くのプログラミング言語で，範囲 $[0, 1)$ に一様に分布する擬似乱数を返す関数が用意されている．図 2.1 は 0 から 9 までのデータが昇順に並んでいる配列 a (◦ 印) と，
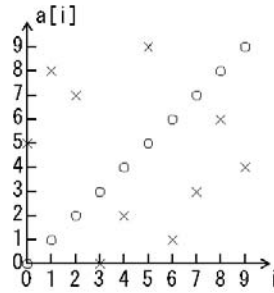
図 2.1　秩序あるデータ ○ 印とシャッフル後の無秩序なデータ × 印．横軸は配列添字の番号であり，縦軸は各配列要素に与えられている値である．（Z02_01_shuffle.java）

乱数を使ってそれをランダムな順番にシャッフルした後の配列 a (× 印) の図である．横軸は配列添字の番号であり，縦軸は各配列要素に与えられている値である．

プログラム (Shuffle.java) のメソッド (shuffle) の一部と，その説明は以下のとおりである．

Shuffle.java のメソッド (shuffle) の一部：

```
 1      for (int j=n-1; j>0; j--) {
 2        im= 0;
 3        rm= r[im];
 4        for (int i=1; i<=j; i++) {
 5          if (rm<r[i]) {
 6            im= i;
 7            rm= r[im];
 8          }
 9        }
10        r[im]= r[j];
11        r[j]= rm;
12        aw= a[im];
13        a[im]= a[j];
14        a[j]= aw;
15      }
```

1. 配列 a の要素には，ある順序でデータが与えられており，a と同じ要素数 n の配列 r の要素には，すでに，範囲 $[0, 1)$ の乱数の値が代入されているとする．
2. 上のプログラムの部分ではその乱数の値をキーとして配列 r を昇順にソートし，それに連動させて対応している a の要素を並べ替える．r の値はランダムに並んでいるから，それをソートすれば，a の値はランダムに並ぶ．
3. L1〜15 [1]: 添字番号が j 以下の r での最大値探しを繰り返すループである．r の最大値に対応する a の値が L14 で a[j] の値となる．
4. L2: im に最大値の添字番号の初期値 0 を代入する．

[1]. プログラムの行番号 n を Ln で表す．

5. L3: rm に最大値の初期値 r[0] を代入する．
6. L4〜9: j 以下で最大値を探すループである．
7. L5〜8: もし rm より大きい値と出会ったら，im，rm の値を i，r[i] の値に変える．
8. L10〜14: 最大値の r[im] と r[j] の要素の入れ替えと，それに連動して a での入れ替えを行う．
9. L15: ループが済むと，配列 r は昇順に並び，配列 a の要素の値は無秩序に並んでいる．

ソートの説明は，2.3 節以降で行う．

2.2　最大値

昇順にソートする例は前節ですでに扱っている．昇順にソートする場合でも，小さな値を探して前から並べる方法と，大きな値を探して後ろから並べる方法が考えられるが，ここでは後者を使うことにする．そこで，一組の n 個のデータをもつ配列 a の中の最大値を求めることから始める．ソートにはいろいろな種類があるように，最大値を求める方法もいろいろ考えられる．最も簡単な方法は，次の方法であろう．

プログラム (Maximum.java) のメソッド (maximumA) の一部と，その説明は以下のとおりである．

Maximum.java のメソッド (maximumA) の一部：

```
1      for (int i=1; i<n; i++) {
2        if (a[i-1]>a[i]) {
3            aw= a[i-1];
4            a[i-1]= a[i];
5            a[i]= aw;
6        }
7      }
```

1. 配列 a の要素にはある無順序のデータが与えられている．
2. 上のプログラムの部分では配列の値をキーとして配列 a を昇順に並べ替える．
3. L1〜7: 配列の最前から順に最大値を探すループである．
4. L2: 現在の配列の値と次の配列の値を比較する．現在の配列の値のほうが大きかったら L3〜5 で入れ替える．
5. L3: a[i−1] の値を aw に退避する．
6. L4: a[i−1] に a[i] の値を代入する．
7. L5: a[i] に退避してある aw の値を代入する．
8. L7: ループが済むと a[n−1] には最大値が代入されている．

この方法を “maximumA” と名付けておく．

最大値を取り出すだけなら，a[i−1]>a[i] の場合にデータの入れ替えをするのは無意味である．そこで，次のように少し改良をしてみる．

Maximum.java のメソッド (maximumB) の一部：

```
1      int imax= 0;
2      double amax= a[imax];
3      for (int i=1; i<n; i++) {
4        if (amax<a[i]) {
5        imax= i;
6        amax= a[imax];
7        }
8      }
```

1. L1: 最大値の添字番号 imax を 0 に初期化する.
2. L2: 最大値を代入する変数 amax を定義し，a[0] を初期値として代入する.
3. L3〜8: 配列の前方から順に最大値を探すループである.
4. L4: 次の配列と比較する.
5. L5, 6: 次の配列がそれまでの最大値より大きかったら，その添字番号を imax に代入し，その値を amax に代入する.
6. L8: ループが済むと amax に最大値が，imax に最大値である a の添字番号が代入されている.

この方法を “maximumB” と名付けておく.

図 2.2 は 0〜9 の値のデータが無秩序に並んでいる配列 a を × 印で，探し出した最大値 amax を座標 (0, amax) に ◦ 印で示した図である.
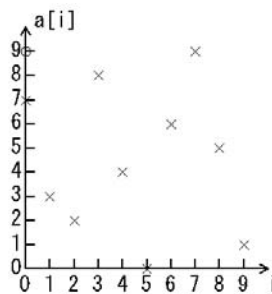
図 2.2　無秩序なデータ × 印と最大値 ◦ 印．横軸は配列添字の番号であり，縦軸は各配列要素に与えられている値である．最大値は座標 (0, amax) の ◦ 印である．(Z02_02_maximum.java)

2.3　バブルソート

バブルソートは，最も単純なソートであり安定なアルゴリズムである．その中でも単純な方法は，“maximumA” を $n-1$ 回繰り返す方法である．図 2.3 は 1〜10 の値がランダムに並んでいる配列 a のデータを，バブルソートを使って昇順に並べ替えてい

i 0 1 2 3 4 5 6 7 8 9
a[i] 4 7 3 6 2 1 5 8 9 10
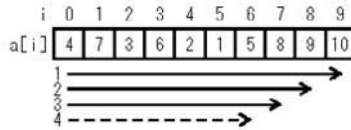
図 2.3　バブルソートの過程．i は配列 a の添字番号，a[i] は要素の値，縦に振ってある数字は最大値を求める何番目の操作かを示す数である．

るところである．1 回目は，a[0] と a[1] の値を比較して，a[0] の値のほうが大きかったら値を入れ替え，順次右に進み a[8] と a[9] の処理を行う．2 回目以降の i 回目では，a[n−i] の後はすでにより大きい値が昇順に並んでいるから，そこまで処理すればよく，最後の n−1 回目は a[0] と a[1] の値の処理で終わらせてよい．図は 4 回目を始めるところであり，a[6] の後はすでにより大きい値が昇順に並んでいる．4 回目は，a[1] の値 7 が右に進み，a[6] の値 5 と入れ替わったら終わることになる．

プログラム (Sort_Bubble.java) のメソッド (sort_Bubble2) の一部と，その説明は以下のとおりである．

Sort_Bubble.java のメソッド (sort_Bubble2) の一部：

```
1     for (int j=n-1; j>0; j--) {
2       for (int i=0; i<j; i++) {
3         if (a[i]>a[i+1]) {
4           aw= a[i];
5           a[i]= a[i+1];
6           a[i+1]= aw;
7         }
8       }
9     }
```

1. L1〜9: 最大値探しを n−1 回繰り返す．
2. L2〜8: 配列の最前から j までで最大値を探す．
3. L3: 現在の配列要素 a[i] の値と直後の要素 a[i+1] の値を比べる．
4. L4〜6: 直後の要素の値より大きかったら入れ替える．
5. L9: ループが終わると，配列 a は昇順になっている．

図 2.4 は 0〜9 の値がランダムに並んでいる配列 a (∘ 印) と，バブルソートを使って昇順に並べ替えた配列 a (× 印) の図である．

この方法では，最前の値からスタートして，大きい値が次々と後へ進んでいく様子が泡が上にボコボコと上がっていく様子に似ている，ということで泡 (bubble) ソートといわれるようである．以上のバブルソートの方法は最も単純であるが無駄もある．プログラム (Sort_Bubble.java) には，他に二つのメソッドがあるから参照してほしい．
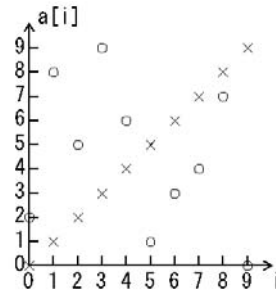
図 2.4　ランダムな配列 ∘ 印とバブルソート後の配列 × 印．（Z02_04_bubble.java）

2.4　櫛ソート

バブルソートの改良版であり，高速であるが不安定である．バブルソートでは，隣り合ったもの同士を比較してソートを行ったが，この方法では，比較する要素の間隔を初めは大きくとり，徐々に狭めていき，最後には，バブルソートと同じように，隣り合ったもの同士を比較してソートを終える．最初の間隔は要素数 n を 1.3 で割った整数とし，繰り返しのたびに間隔を 1.3 で割って，徐々に小さい整数を使う．間隔が小さくなってきたら，11, 8, 6, 4, 3, 2, 1 となるように細工をすると効率が良いことになっている．プログラム (Sort_Comb.java) のメソッド (sort_Comb) の一部と，その説明は以下のとおりである．

Sort_Comb.java のメソッド (sort_Comb) の一部：

```
 1     boolean swd;
 2     double sf= 1.3;
 3     int jh= n;
 4     do {
 5       jh= (int)(jh/sf);
 6       if (jh == 0) jh= 1;
 7       if (jh == 9 || jh == 10) jh= 11;
 8       swd= false;
 9       int i= 0;
10       for (int j=jh; j<n; j++) {
11         if (a[i]>a[j]) {
12           aw= a[i];
13           a[i]= a[j];
14           a[j]= aw;
15           swd= true;
16         }
17         i++;
18       }
19     } while ( jh>1 || swd);
```

1. L1: 入れ替えが起こらなかったら値を false のままにしておくフラグの宣言.
2. L2: 比較するデータの間隔幅を 1.3 で割って狭めていく定数 sf.
3. L3: 比較するデータの間隔幅の初期値であるが，L5 で sf で割られる.
4. L4〜19: 間隔幅を狭めながら，入れ替えを繰り返すための外側のループ.
5. L5: jh の値を狭める.
6. L6: jh=0 になったら swd=false のまま L10〜18 のループが済むまでループを繰り返すために jh=1 とする.
7. L7: jh<=11 (10 または 9) になったら，jh=11, 8, 6, 4, 3, 2, 1 となるようにするために jh=11 とする.
8. L8: 内側のループで，入れ替えが起こらなかったら，そのことを L19 で知るために swd に初期値 false を代入しておく.
9. L9: 比較を行う前方の配列の添字番号を初期値 0 とする.
10. L10〜18: a[0] と比較する後方の配列を a[jh] とし，j が最大の n−1 を超えるまで繰り返すループ.
11. L11: 前方の配列と後方の配列の値の比較を行う.
12. L12〜15: 前方のほうが大きかったら入れ替えを行い，フラグ swd に true を代入する.
13. L17: ループがまわるときに，j だけでなく i の値も 1 増やす必要がある.
14. L19: jh=1 になっており，入れ替えもなく swd=false であったら終了する.

2.5 選択ソート

単純選択法ともいわれ，安定である．最大値を求める "maximumB" の方法の繰り返しである．まず，要素の数が n の配列 a の，a[0] から a[n−1] までで最大値 a[i] を探し，a[i] と a[n−1] の値を入れ替える．次は，a[n−1] には最大値が入っているから，これを抜かして a[0] から a[n−2] までで最大値を探し，a[n−2] の値と入れ替える．この処理を $n-1$ 回繰り返せば終わりとなる．プログラム (Sort_Select.java) のメソッド (sort_Select) の一部と，その説明は以下のとおりである．

Sort_Select.java のメソッド (sort_Select) の一部：

```
 1     for (int j=n-1; j>0; j--) {
 2       imax= 0;
 3       amax= a[imax];
 4       for (int i=0; i<j; i++) {
 5         if (amax<a[i+1]) {
 6           imax= i+1;
 7           amax= a[imax];
 8         }
 9       }
10       a[imax]= a[j];
11       a[j]= amax;
12     }
```

1. L1〜12: n−1 回繰り返される外側のループであり，j は下のループでの最大値を代入する添字番号である．
2. L2, 3: まず，a[0] が最大であると仮定する．
3. L4〜9: 0 から j−1 までで最大値を探すループ．
4. L5: 現在の最大値と次の値を比較する．
5. L6, 7: 次の値のほうが大きかったら，最大値の位置と値を変更する．
6. L10, 11: 内側のループが終わり，その時点での最大値の a と最後の a の値を入れ替える．
7. L12: ループが終了する．

2.6　挿入ソート

単純挿入法ともいわれ，安定である．配列 a の要素の数は n とする．最初に a[1] の値を変数 xw に代入し，a[0] の値と比較する．a[0]>xw であったら a[0] の値を a[1] に移し，xw を a[0] に代入する．次に一つ右の a[2] から始め，まず a[2] を変数 xw に代入して a[1] の値と比較し，a[1]≤xw であったらそこでやめる．そうでなかったら a[1] の値を a[2] に移し，一つ戻って xw と a[0] の値を比較する．a[0]≤xw であったらそこでやめて xw を a[1] に代入し，そうでなかったら a[0] の値を a[1] に移し xw を a[0] に代入する．さらに進んで，a[j] の値を xw に代入し，xw と a[j−1] の値との比較から始めるところまで進んだら，j の値を 1 ずつ減らして調べていく．a[j−1] 以前はすでに昇順に並んでいるから，a[i]<xw となる値をもつ a[i] と出会うまでは a[i] の値を a[i+1] に移し，出会ったらそこでやめてよい．このことを a[n−1] と a[n−2] との比較から始める処理が終わるところまで繰り返せばループが終わりとなる．

図 2.5 は 1〜10 の値のデータが無秩序に並んでいる配列 a を，挿入ソート法を使って昇順に並べ替える過程を示す図である．最初の過程は a[1]=10 と a[0]=5 の比較であり a[0]<a[1] であるから取り替えは行わないので，点線の矢で示してある．2 番目の過程では，a[2] の値 3 が a[0] まで移される．3 番目の過程では，a[3] の値 7 は a[2] ま
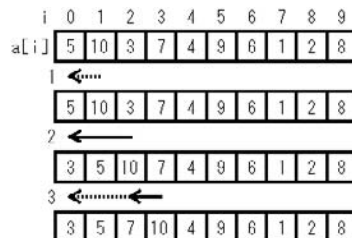
図 2.5　挿入ソートの過程．i は配列 a の要素の番号，a[i] は要素の値，縦に振ってある数字は何番目の操作かを示す数である．実線の矢印は要素の入れ替えのある部分であり，点線の矢印は入れ替えのない部分である．

で移され，a[1] の 5 より大きいので，そこでやめる．挿入ソートという言葉は，この過程における最小値を，より小さな値に出会ったところに挿入するという意味であろう．同じような操作を 9 番目の過程まで行えば終了する．プログラム (Sort_Insert.java) のメソッド (sort_Insert) の一部と，その説明は以下のとおりである．

Sort_Insert.java のメソッド (sort_Insert) の一部：

```
 1      double xw;
 2      for (int j=1; j<n; j++) {
 3        xw= a[j];
 4        for (i=j-1; i>=0; i--) {
 5          if (a[i]>xw) {
 6            a[i+1]= a[i];
 7          } else {
 8            break;
 9          }
10        }
11        a[i+1]= xw;
12      }
```

1. L1: 変数 xw を定義し，これに最小値を代入して，それより大きい値は一つ後へ移す．
2. L2〜12: 最小値を指定する外側のループで，n−1 回繰り返す．
3. L3: j より前は整列しており，次に考慮する最小値の初期値は a[j] である．
4. L4〜10: 最小値とした j より前を調べる．
5. L5: a[i] と現在の最小値 xw を比較する．
6. L6: a[i] のほうが大きかったら a[i] の値を一つ後ろの要素に移す．
7. L8: 最小値より小さい値に出会ったら，それ以前は昇順に整列しているから，そこが今回の最小値の位置となり，ループを終える．
8. L10: L4 のループの終わりに達するか，L8 が実行されるとループが終わる．
9. L11: 最小値を整列の中のしかるべき位置に挿入する．
10. L12: ループが終わると，配列 a は昇順になっている．

2.7　シェルソート

挿入ソートの改良版であり，不安定である．挿入ソートでは，隣り合ったもの同士を比較してソートを行ったが，バブルソートを改良して櫛ソートとしたように，この方法でも，比較する要素の間隔 jh を要素数 n の 1/9 程度の大きさにして挿入ソートを始める．その後，間隔 jh を 1/3 ずつ狭くしてソートを繰り返し，最後は間隔を jh=1 にして単純挿入ソートを行う．昇順のソートは，値の小さなデータを左に，大きなデータを右に移すことであり，シェルソートでは，間隔が jh の大きさで行われるから，全体での処理数が少なくて済むようになる．間隔を開けてソートするので，同じ値の要素の順番が入れ替わってしまうこともあり，この方法は安定ではない．プロ

グラム (Sort_Shell.java) のメソッド (sort_Shell) の一部と，その説明は以下のとおりである．

Sort_Shell.java のメソッド (sort_Shell) の一部：

```
 1    double xw;
 2    int jh= 13;
 3    while (jh<n) jh= 3*jh+1;
 4    jh /= 9;
 5    while (jh>0) {
 6      for (int j=0; j<n-jh; j++) {
 7        xw= a[j+jh];
 8        for (i=j; i>=0; i-=jh) {
 9          if (a[i]>xw) {
10            a[i+jh]= a[i];
11          } else {
12            break;
13          }
14        }
15        a[i+jh]= xw;
16      }
17      jh /=3;
18    }
```

1. L1: 変数 xw を定義し，これに代入する最小値より大きい値は入れ替えて後へ移す．
2. L2: jh の間隔幅ずつ離れたデータを同じ組として処理する．全体を jh 組に分けて，それぞれで挿入ソートを行うことになる．
3. L3, 4: jh の初期値を要素数 n の 1/9 程度の大きさにする．
4. L5〜18: 幅 jh がゼロになるまで繰り返す．
5. L6〜16: 最小値を指定する外側のループで n−jh 回繰り返す．
6. L7: 最小値を指定する．
7. L8〜14: 各組の中で処理するために，−jh ずつずらして繰り返す．
8. L9: a[i] と最小値を比較する．
9. L10: a[i] のほうが大きかったら a[i] の値を同じ組の一つ後ろに代入する．
10. L12: 最小値より小さい値に出会ったら，それ以前は昇順に整列しているので，そこが最小値の位置となり，ループを終える．
11. L14: L8 のループの終わりに達するか，L12 が実行されるとループが終わる．
12. L15: 最小値を整列の中のしかるべき位置に挿入する．
13. L17: 幅 jh を 3 で割り，切り捨てて整数化をする．jh が 1 での処理が済み jh がゼロになるまで繰り返す．
14. L18: ループが終わると，配列 a は昇順になっている．

2.8　クイックソート

クイックソートは一般的には最も高速な方法である．不安定である．メソッドが呼ばれると，まず，データの中で平均値に近い適当な値を閾値 x_m に選ぶ．普通は平均値などはわからないから，配列の中央の要素 a[(l+r)/2] の値を x_m とする．データの左の端の要素から順に x_m より大きい値 x_l をもつ要素 1 個を見つける．次に，データの右の端の要素から順に x_m より小さい値 x_r をもつ要素 1 個を見つけ，それぞれの値 x_l と x_r とを入れ替え，x_m より小さい値は左に，大きい値は右に集まるようにする．再び，左側では l を 1 ずつ増やしながら，x_m より大きい値 x_l を見つけ，右側では r を 1 ずつ減らしながら，x_m より小さい値 x_r を見つけ，x_l と x_r とを入れ替える．その際，l>=r となっていたら，すでに，x_m より大きい値は右側に，x_m より小さい値は左側に集められている．次に，左右両側のデータの各組について，それぞれ，このメソッドの再帰呼び出しを行い，それぞれの要素を大小の二組に分割する．それぞれの組の要素の数が 1 個になったら，再帰呼び出しから戻り，先に進む．すべての組の要素が 1 個ずつになったら，それらは，全体で昇順に並んでいる．プログラム (Sort_Quick.java) のメソッド (sort_Quick) の一部と，その説明は以下のとおりである．

Sort_Quick.java のメソッド (sort_Quick) の一部：

```
 1   public void sort_Quick(final int l, final int r,
 2       double[] a) {
 3     double xm= a[(l+r)/2];
 4     double aw;
 5     int lw= l, rw= r;
 6 //  閾値より小さい値は左へ，大きい値は右へ並べる
 7     for ( ; ; ) {
 8       while (a[lw]<xm) lw++;
 9       while (xm<a[rw]) rw--;
10       if (lw<rw) {
11         aw= a[lw];
12         a[lw]= a[rw];
13         a[rw]= aw;
14         lw++;
15         rw--;
16       } else {
17         break;
18       }
19     }
20     if (l<lw-1) sort_Quick(l, lw-1, a);
21     if (rw+1<r) sort_Quick(rw+1, r, a);
22   }
```

1. L1, 2〜22: メソッド sort_Quick の定義である．配列 a と，その中でソートする部分の左端 l と右端 r を引数とする．
2. L3: データを左右に分ける閾値として中央近くの値を指定する．

3. L5: 左端と右端の添字番号を変数 lw と rw にコピーする.
4. L7〜19: 閾値より小さい値は左に, 大きい値は右に並べるループであり, 仕分けが済むと L17 の break 文でループを抜け出す.
5. L8: 左端から調べ, 閾値より小さければ変数 lw に 1 を加えることを繰り返し, 閾値より大きければ右側に移すために次に進む.
6. L9: 右端から調べ, 閾値より大きければ変数 rw から 1 を引くことを繰り返し, 閾値より小さければ左側に移すために次に進む.
7. L10: 左端変数 lw と右端変数 rw を比較する.
8. L11〜15: 左端変数 lw が右端変数 rw より小さければ, 処理は済んでいないから, 左側の大きい値と右側の小さい値を入れ替え, 変数値 lw と rw の値を 1 ずつ増減する.
9. L17: 左端変数 lw が右端変数 rw より小さくなければ, 処理が済んでいるので, ループを終える.
10. L20: 左側にデータがまだ複数あれば再帰的にソートをする.
11. L21: 右側にデータがまだ複数あれば再帰的にソートをする.
12. L22: 再帰呼び出しでの終了の場合には呼び出し元に戻り, 最初の呼び出しの処理でここまで来ると, すべてのソートが終わっている.

図 2.6 は 1〜10 の値のデータが無秩序に並んでいる配列 a の値を, クイックソートによって昇順に並べ替える過程を示す図である. 最初の過程は $x_m = 4$ が閾値であるから, まず, 左端の 5 と右から 2 番目の 2 が入れ替わる. 続いて 10 と 1, 最後に 7 と 4 が入れ替わる. 2 番目の過程で, 左側の部分では $x_m = 1$ であるから, 2 と 1 が入れ替わるだけで終わり, 右側の部分の処理に進む前に左側で先へ進む. 3 番目の過程で, 左端の部分の要素は 1 個しかないから, そのままで右隣の部分の処理に移る. この部分の処理が全部済むと, 2 行目の右半分に進み, $x_m = 6$ を閾値として処理を行う. 同様の操作を繰り返して, 1 行目に戻ってくるまで処理を行う.

図 2.6　クイックソートの過程. i は配列 a の添字番号, a[i] は要素の値である. 行の途中にある数値は, データを左右に分ける閾値 x_m である. 縦の実線は, 配列が 2 分割されたときの境である.

2.9　　ヒープソート

この方法も高速であるが，不安定である．図2.7の左右の図は，実際の木とは上下が逆であるが，それぞれが木構造(tree)をなしているといわれる．左図で①は根(root)，②～⑤は節(nod)，⑥～⑩は下につながらず先端であるから葉(leaf)，これらを結んでいる線は枝(branch)といわれる．各節は，上にある根や節の子であり，下にある節や葉の親である．この図のように，根や節の下で2本の枝に分かれている木は2分木(2進木)といわれる．

左図の数字は配列添字の番号である．添字番号を0からではなく1から始めると，番号がiの節の子の番号は$2i$と$2i+1$となり，逆に子の番号jが偶数でも奇数でも，その親の番号は$j/2$となる(単純に整数化するときは，小数は切り捨てられる)ので，親子関係が明解になる．そこで，説明文では添字番号を1から始め，javaでは添字番号を0から始めることにする．

右図の数字は各配列要素の値であり，1～10の値が代入されている．ヒープ(heap)とは堆積であるが，ヒープソートの場合にはすべての根あるいは節について，それぞれの子の値が自分の値より小さくなるように配置されている場合に，ヒープを構成しているといわれる．

ヒープソートの処理は，大別すると二つの処理に分けられる．

❒ 第一は，全データ(右図)を使ってヒープを作ることである．ヒープを作る処理の最小の単位処理は，親と左右の子のデータ3個(右側の子がない場合は2個)のうち最大のデータを親のデータとすることである．配列aの全データ$x_1, \cdots, x_n$を使ったヒープを作るには，親である(葉ではない)最後の配列要素$a_{n/2}$(左図の⑤)から単位処理を行う．それが済むと，この場合，子は葉であるから孫はなく，親$a_{n/2}$から下はヒープとなっている．以後，親にする配列要素の添字番号を1ずつ減らしながら単位処理を行うが，親と子のデータの入れ替えがあり，しかも孫がある場合には，親のデータが代入された子から下は，ヒープを構成しているとは限らないので，それを親として下へ単位処理を繰り返していく必要がある．最後に親がa_1の場合の処理が済めば，
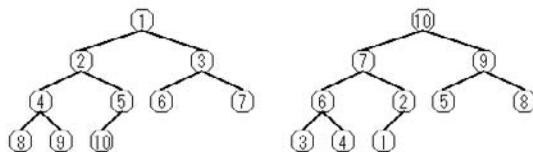
図2.7　ヒープソートの木構造．要素数が10の配列を扱う．左図の数字は配列添字の番号である．右図の数字は各配列要素の値であり，1～10の値が代入されている．この右図の数値をソートすることになる．

全データがヒープを作っている．

❒ ヒープが完成すると，根の位置 a_1 には必ず最大値 (右図の 10) があるから，それと，配列の最後の要素 a_{10} の値 (右図の 1) とを入れ替える．最大値が配列の最後の要素に移り，最大値でないものが根の位置に移る．そこで，配列の最後の要素を除外した要素全体 (右図で 9 個) で a_1 を親としてヒープを作りなおし，再び，根と配列の最後の要素 a_9 とを入れ替える．この処理を 2 個の要素だけでの処理が済むまで繰り返すと，配列の要素全体が昇順に並んでおり，ソートが終了する．

プログラム (Sort_Heap.java) のメソッド (sort_Heap) の二つの部分と，それぞれの説明は以下のとおりである．

Sort_Heap.java のメソッド (sort_Heap) の一部：

```
 1     for (i=n/2; i>=1; i--) {
 2       heap(i, n, a);
 3     }
 4     while (n>1) {
 5       n--;
 6       aw= a[n];
 7       a[n]= a[0];
 8       a[0]= aw;
 9       heap(1, n, a);
10     }
```

1. L1〜3: 全データのヒープを構成するために，最後の親から根までの部分的なヒープを繰り返し作っていく．最後の親は n/2 番目の要素 a[n/2−1] ($= a_{n/2}$) である．
2. L2: i 番目の要素を親として，それ以下 n 番目の要素までで必要な要素を使って，部分的なヒープを作るメソッド heap を呼ぶ．
3. L3: 根から下のヒープが完成したらループを抜ける．
4. L4〜10: ソートを行う．ソートされていない要素の数が 1 になる (ソートが済む) までループを繰り返す．
5. L5: 現時点での最後の要素以後はソートされたので，次の処理からはずすため，要素数 n を 1 減らす．
6. L6〜8: 根 a[0] と現時点での最後の要素 a[n] の値を入れ替える．次の処理は a[0]〜a[n−1] の n 個で行う．
7. L9: その最後の要素 a[n] 以前でヒープ作りを繰り返す．
8. L10: 残された要素の数が 1 個になると，全体がソートされているから，ソートが終了する．

Sort_Heap.java のメソッド (heap)：

```
 1   public void heap(int i, int n, double[] a) {
 2     double aw= a[i-1];
 3     int j=i*2;
 4     while (j <= n) {
```

```
 5        if (j<n && a[j-1]<a[j]) {
 6          j++;
 7        }
 8        if (a[j-1]>aw) {
 9          a[i-1]= a[j-1];
10          i = j;
11          j= i*2;
12        } else {
13          break;
14        }
15      }
16      a[i-1]= aw;
17    }
```

1. L1〜17: メソッド heap の定義である．引数の配列 a の i 番目から n 番目までのデータでヒープを作る．
2. L2: 処理の先頭の i 番目の要素を親とする．
3. L3: 左側の子 (左子) は i*2 番目である．
4. L4〜15: 子の番号が n を超えていたら終わる．
5. L5: 右子がおり，その値が左子の値より大きければ，
6. L6: 右子を親と比較するため j の値を 1 増やして，右子の番号にする．
7. L8: 大きいほうの子の値と親の値を比較する．
8. L9: 親より大きければ親にその子の値を代入する．
9. L10: 次に，これまでの左子を親にしてさらに子孫に進む．
10. L11: 新しい左子の番号を j に代入する．
11. L13: 親が子より大きかったら，それ以下はすでにヒープを形成しているから while のループを抜ける．
12. L15: 子の番号が n を超えたり，親が子より大きかったらループが終わる．
13. L16: 最後の親に最初の親の値を代入してヒープが完成する．
14. L17: メソッドを終える．

2.10　マージ

ソートでは，乱雑に並んでいる一組のデータをもつ 1 個の配列を処理して，ある規則に従ってデータの整列を作っている．他方，データがすでに整列している複数の配列を，1 個の配列に整列したデータとして合併することをマージ (併合) という．図 2.8 の ∘ 印の左 5 個は配列 a であり，右 5 個は配列 b である．図の × 印は，配列 a と b をマージした配列 c である．

プログラム (Merge.java) のメソッド (merge) の一部と，その説明は以下のとおりである．
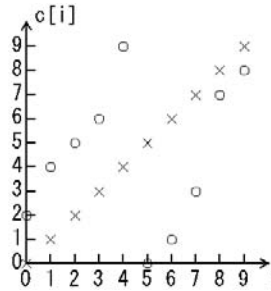
図 2.8　2 個の配列のマージ．○ 印の左 5 個は配列 a，右 5 個は配列 b，× 印は配列 a と b をマージした配列 c である．（Z02_08_merge.java）

Merge.java のメソッド (merge) の一部：

```
 1      int ia=0, ib=0, ic=0;
 2      while (ia<na && ib<nb) {
 3        if (a[ia] <= b[ib]) {
 4          c[ic++]= a[ia++];
 5        } else {
 6          c[ic++]= b[ib++];
 7        }
 8      }
 9      while (ia<na) c[ic++]= a[ia++];
10      while (ib<nb) c[ic++]= b[ib++];
```

1. L1: 配列 a，b，c の添字番号をゼロに初期化する．
2. L2〜8: 配列 a, b のどちらかで，添字の値が要素数を超えたら，ループを終了し L9 へ移る．
3. L3: 配列 a の要素の値 a[ia] と配列 b の要素の値 b[ib] を比較する．
4. L4: a[ia] のほうが大きくなかったら，a[ia] を配列要素 c[ic] に代入して ia，ic を 1 増やす．
5. L6: a[ia] のほうが大きかったら，b[ib] を配列要素 c[ic] に代入して ib，ic を 1 増やす．
6. L8: どちらか片方の要素が終わったらループを抜ける．
7. L9, 10: まだ要素が残っているほうの要素を c に追加コピーして終了する．

図 2.9 は整列した配列 a と b を配列 c にマージする様子を示している．初めに a[0] の 2 と b[0] の 0 を比較し，小さいほうの 0 を c[0] に代入する．次は，残された a[0] の 2 と b[1] の 1 を比較し，小さいほうを c に代入する．このことを繰り返して，b[4] の 6 と a[2] の 7 を比較し，小さいほうの 6 を c[6] に代入した後，b の要素は終わる．その後は a の残された要素を c にコピーするだけである．
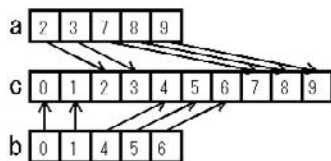
図 2.9　2 個の配列のマージの説明．配列 a と b は，整列した配列である．c は a と b をマージした配列である．

2.11　マージソート

原理は，次のとおりである．整列していない配列 a がある．全データをほぼ中央で 2 分割して 2 組に分ける．さらに，左右の 2 組をそれぞれ 2 分割して 4 組に分ける．以下，1 組の要素の数が 1 個になるまで 2 分割を続ける．要素の数が 1 個になった組は，それだけで整列していると見なせる．要素の数が 1 個になった組に到達したら，今度は 2 分割の過程を逆にたどって，2 組ずつをマージしていけば，最後には，全体が整列した配列になる．

図 2.10 は配列 a の 2 分割を繰り返して，要素が 1 個ずつになる様子である．まず，a を b と c の部分に 2 分割するが，b，c 等は説明のために付けた配列 a の部分配列の名前である．

プログラム (Sort_MergeSort.java) の二つのメソッド (sort_MergeSort と merge)，また，それぞれの説明は以下のとおりである．

Sort_MergeSort.java のメソッド (sort_MergeSort)：

```
1  public void sort_MergeSort(int l, int r, double[] a) {
2    if(l<r){
3      int m= (l+r-1)/2;
```
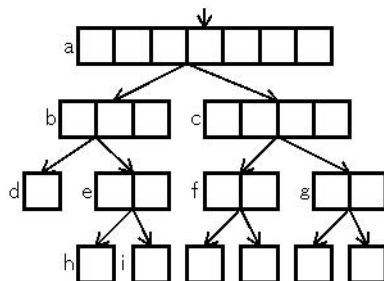
図 2.10　マージソートの説明．a は元の配列である．b と c の部分に 2 分割し，さらに，それぞれを 2 分割して，最後には要素 1 個ずつになる．

```
4       sort_MergeSort(l, m, a);
5       sort_MergeSort(m+1, r, a);
6       merge(l, r, a);
7     }
8   }
```

1. L1〜8: メソッド sort_MergeSort の宣言であり，配列 a の添字番号の l 番目から r 番目までのデータをソートする．
2. L2: l=r で要素が 1 個になったら何もせずに戻る．
3. L3: 左右の分割点 m を指定．
4. L4: 左側をソートする．m は左側に属することになる．
5. L5: 右側をソートする．
6. L6: ソートされた左右をマージする．
7. L8: ソートが終了する．

Sort_MergeSort.java のメソッド (merge)：

```
1   private void merge(int l, int r, double[] a){
2     int i;
3     double[] aw= new double[a.length];
4     int m= (l+r-1)/2;
5     int ill= l;
6     int irl= m+1;
7     int k= l;
8     while (ill<=m && irl<=r){
9       if (a[ill]<a[irl]) {
10        aw[k++]=a[ill++];
11      } else{
12        aw[k++]= a[irl++];
13      }
14    }
15    while (ill<=m) aw[k++]= a[ill++];
16    while (irl<=r) aw[k++]= a[irl++];
17    for (i=l; i<=r; i++) {
18      a[i]= aw[i];
19    }
20  }
```

1. L1〜20: メソッド merge の宣言である．引数の配列 a の添字番号の l 番目から L4 で与えられる m 番目までと，その次から r 番目までのデータをマージする．
2. L3: マージするときにマージ先とする作業用配列 aw の宣言．
3. L4: マージする配列 a の分割位置であり，m は左側に属する．
4. L5〜7: マージする左側と右側の左端用変数と，作業用配列の左端用変数に値を代入する．
5. L8〜14: 両側とも残りがある間はマージを繰り返す．
6. L9〜13: 左右の値を比較して，右側が大きければ左側を作業用配列にコピーし，大きくなければ右側を作業用配列にコピーする．
7. L15, 16: 左側の残りがあれば作業用配列にコピーし，右側の残りがあれば作業用配列にコピーする．

8. L17〜19: 作業用配列の値の l から r までを，元の配列 a にコピーする．

このことを図 2.10 でたどることにする．a を 2 分割して b と c にする．b を 2 分割して d と e にする．d は要素が 1 個であるから，すでにソートされていると考えて，b に戻り，e に進む．e を 2 分割して h と i にする．h と i は要素が 1 個であるから，h と i では何もせずに e に戻り，h と i のマージを行う．b に戻り d と e のマージを行い a に戻る．次に，右側の処理をするために c に進むが，戻ってきたら b と c のマージをして，ソートが終了する．

演習問題

[2.1] プログラム Maximum.java，MaximumTest.java を修正して，最小値を求めるプログラム Minimum.java , MinimumTest.java を作りなさい．

解答例： ソースプログラム Minimum.java，MinimumTest.java

[2.2] プログラム Sort_Bubble.java，Sort_BubbleTest.java を修正して，降順にソートするプログラム Sort_Bubble_d.java，Sort_Bubble_dTest.java を作りなさい．

解答例： ソースプログラム Sort_Bubble_d.java，Sort_Bubble_dTest.java

[2.3] プログラム Sort_Quick.java，Sort_QuickTest.java を修正して，降順にソートするプログラム Sort_Quick_d.java，Sort_Quick_dTest.java を作りなさい．

解答例： ソースプログラム Sort_Quick_d.java，Sort_Quick_dTest.java

第 3 章

補間法

少数の離散的なデータの組 (x_i, y_i)（ただし $0 \le i \le n$）が与えられているとき，すべての x_i に対して $f(x_i) = y_i$ が成り立つ関数で，任意の x に対する関数値 $y = f(x)$ を求める方法を補間法といい，$y = f(x)$ を補間式という．$f(x)$ を多項式として，その各項の係数をデータ点の値を使って定めた式を補間多項式といい，特に，スプライン関数を使う場合はスプライン補間式という．x がデータとして与えられた点 x_i 等のいくつかに挟まれた範囲にある場合には内挿といい，そうでない場合には外挿という．一般的に，外挿は誤差が大きくほとんど役に立たない．

3.1 ラグランジュの補間法

補間多項式として 2 次式を使う場合を検討しよう．2 次式は

$$y = f_2(x) = ax^2 + bx + c \tag{3.1}$$

である．この関数は 3 個のパラメータ a, b, c を使っているから，これらの値を決めるためには 3 組のデータが必要になる．3 組のデータとして，補間値を求めようとする x に近い 3 点を使うと誤差を小さくすることができる．それらを x_0, x_1, x_2 とする．求める補間式 (3.1) に 3 点のデータ値を代入すると，

$$\left.\begin{aligned} y_0 &= ax_0^2 + bx_0 + c \\ y_1 &= ax_1^2 + bx_1 + c \\ y_2 &= ax_2^2 + bx_2 + c \end{aligned}\right\} \tag{3.2}$$

となる．ここで，未知数であるパラメータ a, b, c の解を求めると

$$\left.\begin{aligned} a &= \frac{(y_0x_1 + y_1x_2 + y_2x_0) - (y_0x_2 + y_1x_0 + y_2x_1)}{(x_0^2x_1 + x_1^2x_2 + x_2^2x_0) - (x_0^2x_2 + x_1^2x_0 + x_2^2x_1)} \\ b &= \frac{(x_0^2y_1 + x_1^2y_2 + x_2^2y_0) - (x_0^2y_2 + x_1^2y_0 + x_2^2y_1)}{(x_0^2x_1 + x_1^2x_2 + x_2^2x_0) - (x_0^2x_2 + x_1^2x_0 + x_2^2x_1)} \\ c &= \frac{(x_0^2x_1y_2 + x_1^2x_2y_0 + x_2^2x_0y_1) - (x_0^2x_2y_1 + x_1^2x_0y_2 + x_2^2x_1y_0)}{(x_0^2x_1 + x_1^2x_2 + x_2^2x_0) - (x_0^2x_2 + x_1^2x_0 + x_2^2x_1)} \end{aligned}\right\} \tag{3.3}$$

である．これを式 (3.1) に代入すれば 2 次式が定められる．それを変形して整理すると

$$f_2(x) = y_0\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} + y_1\frac{(x-x_2)(x-x_0)}{(x_1-x_2)(x_1-x_0)} + y_2\frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} \tag{3.4}$$

となる．この式は書き換えると，

$$f_2(x) = \sum_{i=0}^{2}\left(y_i \prod_{j=0\,(j\neq i)}^{2} \frac{x-x_j}{x_i-x_j}\right) \tag{3.5}$$

である[1]．n 次式に拡張すると，

$$f_n(x) = \sum_{i=0}^{n}\left(y_i \prod_{j=0\,(j\neq i)}^{n} \frac{x-x_j}{x_i-x_j}\right) = \sum_{i=0}^{n} y_i\Lambda_i(x) \tag{3.6}$$

となることは，容易に想像できるであろう[2]．実際，データ点の一つの x_k を式 (3.6) に代入すると，$\sum$ の項の中で $i=k$ 以外の項はゼロとなり，$i=k$ の項は y_k となるから $f(x_k)=y_k$ が成り立つので，この式は確かに補間公式である．これらは，ラグランジュの補間公式といわれる．式 (3.6) を見るとわかるように，データ点 (x_0，x_1，$\cdots$，x_n) は，大きさの順に並んでいる必要はないし，間隔も一様でなくてよい．しかし，いろいろな値の x について精度が一様になるようにするために，等間隔にとったり，あるいは，値の変化の激しい領域では間隔をより小さくとったりしている．

次に，この式の精度を評価してみる．真の式を $f(x)$ とすれば，p を一つのパラメータとして

$$f(x) = f_n(x) + p\Pi_n(x) + R_{n+2}(x) \tag{3.7}$$

と置くことができる[3]．ただし，関数

$$\Pi_n(x) = \prod_{i=0}^{n}(x-x_i) \tag{3.8}$$

は $(n+1)$ 次式，$R_{n+2}(x)$ は $(n+2)$ 次以上の式であり，すべてのデータ点 $x=x_i$ ($i=0$，$\cdots$，n) で $R_{n+2}(x)=0$ である．ここで，p をすべてのデータ点とは異なる点 x' で $f(x') = f_n(x') + p\Pi(x')$ が成り立つような p' とする．p をこのように選ぶと $R_{n+2}(x')$ もゼロとなる．この p' を式 (3.7) に代入して，その両辺の $(n+1)$ 階微分をとると，

$$f^{(n+1)}(x) = p'\Pi^{(n+1)}(x) + R_{n+2}^{(n+1)}(x) \tag{3.9}$$

---

1. $\sum$ は直後の項の和（サム）であり，$\prod$ は直後の項の積（プロダクト）である．$(j\neq i)$ は j が i に等しくない場合に限定している．
2. $\Lambda_i(x)$ は関数である．
3. $\Pi_n(x)$ は関数であり，項の積を表す記号 $\prod$ ではない．

である．さらに $\Pi^{(n+1)}(x) = (n+1)!$ である．$R_{n+2}(x)$ は点 x' と x_i $(i = 0, \cdots, n)$ の合計 $(n+2)$ 点でゼロとなるから，$R_{n+2}^{(n+1)}(x)$ も x'，x_i $(i = 0, \cdots, n+1)$ の $(n+2)$ 点の範囲内でゼロとなる点 x'' が一つは必ずある．その点 x'' を使えば，式 (3.9) から

$$p' = \frac{f^{(n+1)}(x'')}{(n+1)!} \tag{3.10}$$

が決まり，点 x' における誤差は

$$f(x') - f_n(x') = \frac{f^{(n+1)}(x'')}{(n+1)!}(x' - x_0)\cdots(x' - x_n) \tag{3.11}$$

である．$f^{(n+1)}(x)$ がほとんど一定であるなら，一般的に誤差は

$$f(x) - f_n(x) \approx \frac{f^{(n+1)}(x)}{(n+1)!}(x - x_0)\cdots(x - x_n) = \frac{f^{(n+1)}(x)}{(n+1)!}\Pi_n(x) \tag{3.12}$$

であるといえよう．そこで，$(n+1)$ 個以上のデータがある場合には，精度を上げるためには $(x - x_0)\cdots(x - x_n)$ で使うデータ点 $x_0, \cdots, x_n$ を，x がその分布の中央にくるように選べばよいことになる．

プログラム (Lag.java) のメソッド (lag) の一つと，その説明は以下のとおりである．

Lag.java のメソッド (lag) の一つ：

```
1   public double lag(double x, double[] xd, double[] yd) {
2     int n= xd.length;
3     double[] pr= new double[n];
4     double w1, w2;
5     for (int i=0; i<n; i++) {
6       w1= w2= 1.0;
7       for (int j=0; j<n; j++) {
8         if (i == j) continue;
9         w1 *= x-xd[j];
10        w2 *= xd[i]-xd[j];
11      }
12      pr[i]= w1/w2;
13    }
14    double y= 0.0;
15    for (int i=0; i<n; i++) {
16      y += yd[i]*pr[i];
17    }
18    return y;
19  }
```

1. L1〜19: メソッド lag の定義である．引数は補間値を求める x，既知のデータ点 (x, y) の配列 xd，yd である．
2. L3: 式 (3.6) のプロダクト項 $\Pi\dfrac{(x - x_j)}{(x_i - x_j)}$ の配列．
3. L4: $\Pi(x - x_j)$，$\Pi(x_i - x_j)$ の作業変数．
4. L5〜13: プロダクト項を n 項作る．
5. L6: $\Pi(x - x_j)$，$\Pi(x_i - x_j)$ の作業変数の初期化．

6. L7〜11: $\Pi(x-x_j)$, $\Pi(x_i-x_j)$ の作成.
7. L8: プロダクト項の作成で i=j の項は抜かす.
8. L9: $\Pi(x-x_j)$ の作成.
9. L10: $\Pi(x_i-x_j)$ の作成.
10. L12: $\Pi\dfrac{(x-x_j)}{(x_i-x_j)}$ の作成.
11. L14: 補間値の初期化.
12. L15〜17: n 項の積和を作る.
13. L16: 積和.
14. L18: 補間値を戻す.

図 3.1 は指数関数 e^x の補間の精度が，データ点の数によってどのように変わるかを示したものである．データ点の範囲は $[-2, 2]$ であり，それを 1〜5 等分して作ったデータを使って，それぞれ 1〜5 次式で補間をしている．1 次式の場合は，データ点は -2, 2 の 2 点であり，補間式は実線の直線である．2 次式の場合は，データ点は -2, 0, 2 の $+$ 印 3 点であり，補間式は点線の 2 次式である．3 次式は $*$ 印 4 点と破線，4 次式は $\circ$ 印 5 点と鎖線，5 次式は $\times$ 印 6 点と一点差線である．4 次式の鎖線と 5 次式の一点差線はほぼ重なっており，図で見る精度では収束している．

表 3.1 は図 3.1 と同じ補間の $x=0$ における相対誤差である．ただし，$x=0$ がデータ点とならないような，次数が偶数 (データ点が奇数) の場合のみを表に載せてある．次数が 1 増すごとに精度が 1 桁弱良くなっている．

式 (3.12) を使って $x=0$ での誤差の検討をしてみる．指数関数は何階微分を行っても関数形が変わらず $f^{(n+1)}(0)=1$ である．データ点の範囲 $[-2, 2]$ は共通であり，次数が増えるとその範囲内でデータ点の数が増すので，$\Pi_n(0)$ は表 3.1 からわかるように，次数が増してもあまり小さくならない．他方，分母の $(n+1)!$ は n が 10 以上に大きくなると，次数が 1 次増えると 1 桁以上大きくなる．このことは，表の数値からも読み取ることができる．当然なことであるが，精度を上げるためには，$\Pi_n(x)$ の値が小さくなるように，刻みを細かくし，使用するデータ点の x 座標の範囲を小さくすれ
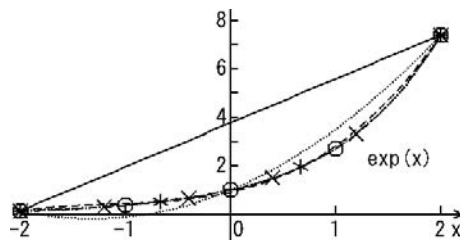
図 3.1　ラグランジュの補間法におけるデータ点の数と精度の関係．関数は指数関数 e^x，データ点の範囲は $[-2, 2]$，実線の直線はデータ点，-2, 2 の 2 点を使った 1 次式での補間である．$+$ 印 3 点と点線は 2 次式，$*$ 印 4 点と破線は 3 次式，$\circ$ 印 5 点と鎖線は 4 次式，$\times$ 印 6 点と一点差線は 5 次式である．(Z03_01_lagrange.java)

表 3.1　ラグランジュの補間法におけるデータ点の数と精度の関係．関数は指数関数 e^x，データ点の範囲は $[-2, 2]$ である．第 1 列と第 4 列は補間式 $f(x)$ の次数，相対誤差は $x = 0$ における値 $\{f(0) - 1\}$ であり，$\Pi_n(0)$ は誤差の評価の式 (3.12) にある項である．$x = 0$ がデータ点とならないような，次数が偶数 (データ点が奇数) の場合のみを表に載せてある．(H03_01_lagrange.java)

次	相対誤差	$\Pi_n(0)$	次	相対誤差	$\Pi_n(0)$
2	−2.762195	−4.0	10	−7.687E−8	−0.2622
4	0.085876	1.7777	12	3.10E−10	0.1410
6	−0.001414	−0.9216	14	−9.11E−13	−0.0759
8	1.310E−5	0.4895	16	2.10E−15	0.0409

ばよい．

図 3.2 はデータ点のとり方によって，精度がどのように変わるかを示すもう一つの例である．データとして，範囲 $[-10.5, 10.5]$ を 21 等分した 22 点の座標 x と指数関数 $\exp(x)$ の値を用意する．補間値を求める点は $x = 0$ として，使用するデータ点は先に用意したデータ点のみとする．$x = 0$ はデータ点ではない．$x = 0$ における真値は $\exp(0.0) = 1.0$ である．図の横軸は補間に使った多項式の次数 n であり，その際使用したデータ点の数は $n + 1$ である．縦軸は補間により求められた補間値である．$\circ$ 印は使用するデータ点を補間値を求める点 $x = 0$ を挟んで左右ほぼ同数とった場合の補間であり，$(n + 1)$ が奇数の場合，例えば $n = 2$ では左側に 2 点，右側に 1 点である．$\times$ 印は $x = 0$ の左側に常に 1 点，その他は右側を増やしていった場合のものである．前者でも真値への収束は遅く，後者では収束せずに発散が始まっている．収束の遅い原因は，データ点の数を増やすときに刻みを細かくするのではなく，より外側の点を使うので式 (3.12) の項 $\Pi_n(x)$ が大きくなるという一般的な理由からである．このように，より遠くのデータまで使うことでは精度を上げられないことがある．
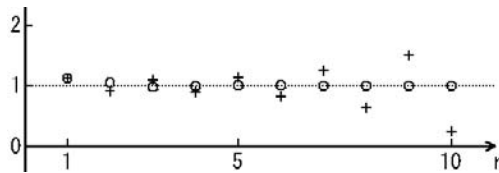
図 3.2　ラグランジュの補間法におけるデータ点のとり方と精度の関係．横軸は補間に使った多項式の次数，縦軸は補間により求められた補間値である．詳細は本文を参照のこと．(Z03_02_lagrange.java)

3.2 ラグランジュ補間の汎用プログラム

ラグランジュの補間多項式は前節で述べたとおりであるが，実用的に使いやすいプログラムが必要である．それは，x のある範囲内で昇順あるいは降順の $n+1$ 点のデータの組 $(x_0, y_0), \cdots, (x_n, y_n)$ を与え，補間点 x と補間に使用するデータ点の数 m を指定すると，点 x に隣接した $m+1$ 個のデータを使って，m 次のラグランジュ補間で求めた答えを戻すプログラムである．図 3.3 は関数 $\sin(x)/x$ のデータ値として，$[-6\pi, 6\pi]$ を 30 等分したデータ 31 組を与え，それを補間する 3 次のラグランジュ補間で，-6.5π から 6.5π までの多くの x 点で補間値を求め，グラフにしたものである．$\circ$ 印は与えたデータ点である．$|x| \leq 6\pi$ の実線は内挿補間であるから近似が良いが，$|x| > 6\pi$ の領域での点線部分は外挿になっているので，両端のデータ点から離れるに従って近似が悪くなっていく．

ラグランジュの補間法によると，微分値を求めることも可能であり，式 (3.6) を微分した式

$$f_n'(x) = \sum_{i=0}^{n} y_i \Lambda_i'(x) \tag{3.13}$$

を使えばよい．ただし，

$$\Lambda_i'(x) = \sum_{k=0\,(k\neq i)}^{n} \frac{1}{(x_i - x_k)} \prod_{j=0\,(j\neq i,\,k)}^{n} \frac{x - x_j}{x_i - x_j} \tag{3.14}$$

である．しかし，x の値が変わり，使うデータの区分が移ると補間式自体が変わるから，そのデータ点では微分値が連続にならないので注意が必要である．図 3.3 の破線と一点差線は補間により求めた微分値であり，$\times$ 印は関数の微分から求めた真値である．

プログラム (Lagrange.java) のメソッド (lagrange) と，その説明は以下のとおりである．
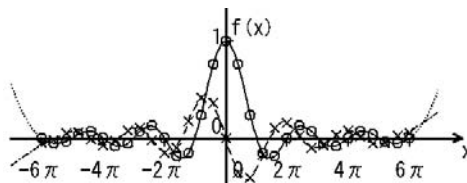
図 3.3　ラグランジュの補間法の汎用プログラムによる出力．関数は $\sin(x)/x$ であり，$\circ$ 印は与えたデータ，実線は補間曲線である．$|x| > 6\pi$ の領域での点線部分は外挿である．破線は補間により求めた微分曲線であり，$\times$ 印は関数の微分から求めた真値である．$|x| > 6\pi$ の領域での一点鎖線部分はその外挿である．（Z03_03_lagrange.java）

Lagrange.java のメソッド (lagrange)：

```
 1   public double lagrange(double x, double[] xd, double[] yd,
 2               int min, int max, int n) {
 3     int i, j;
 4     double[] pr= new double[n+1];
 5
 6     if (max-min <= n) {
 7       System.out.println("Error: n="+n+" is largeer than max-min.");
 8       return 0;
 9 //      System.exit(-1);
10     }
11
12     int imin,imax;
13     int n1= (n+1)/2;
14     int n2= (n+2)/2;
15     double s= xd[max]-xd[min];
16
17     imin= max-n;
18     for (i=min+n2; i<=max-n1; i++) {
19       if ((x-xd[i])*s < 0) {
20         imin= i-n1;
21         break;
22       }
23     }
24     if ((n/2)*2==n && Math.abs(x-xd[i-1])<Math.abs(x-xd[i])) {
25       imin--;
26     }
27     imax= imin+n;
28
29     double w1, w2;
30     for (i=imin; i<=imax; i++) {
31       w1= w2= 1.0;
32       for (j=imin; j<=imax; j++) {
33         if (i == j) continue;
34           w1 *= x-xd[j];
35           w2 *= xd[i]-xd[j];
36       }
37       pr[i-imin]= w1/w2;
38     }
39
40     double y= 0.0;
41     for (i=imin; i<=imax; i++) {
42       y += yd[i]*pr[i-imin];
43     }
44     return y;
45   }
```

1. L1, 2〜45: メソッド lagrange の定義である．引数の x は補間値を求める x 座標，xd, yd は既知のデータ点 (x, y) の配列，min は使用する配列の最小添字番号，max は使用する配列の最大添字番号であり，n は補間式の次数である．戻

り値は補間値である.

2. L4: プロダクト項 $\Pi\dfrac{(x-x_j)}{(x_i-x_j)}$ の配列.
3. L6〜10: 次数 n ≤ max − min でない場合に警告を出す.
4. L15: 変数 s に xd[max] − xd[min] を代入し昇順なら正，降順なら負とする．x 座標は昇順でも降順でもかまわない.
5. L17〜26: 内挿の場合は使うデータ点の中央近くに x が来るようにデータ点を選び，外挿の場合は x に近い点から選んで，補間に使う最小添字番号を決定する.
6. L27: 補間に使う最大添字番号を決定する.
7. L29〜38: プロダクト項 $\Pi\dfrac{(x-x_j)}{(x_i-x_j)}$ の作成.
8. L40〜43: 補間値の計算.
9. L44: 補間値を返す.

プログラム (Lagrange.java) の微分値を求めるメソッド (d_lagrange) の一部と，その説明は以下のとおりである.

Lagrange.java のメソッド (d_lagrange) の一部：

```
 1  public double d_lagrange(double x, double[] xd, double[] yd,
 2              int min, int max, int n) {
 3  中略
 4    double w1, w2, w3;
 5    for (i=imin; i<=imax; i++) {
 6      w1= 1.0;
 7      w2= 0.0;
 8      for (k=imin; k<=imax; k++) {
 9        if (k == i) continue;
10        w3 = 1/(xd[i]-xd[k]);
11        for (j=imin; j<=imax; j++) {
12          if (j==i || j==k) continue;
13          w3 *= (x-xd[j])/(xd[i]-xd[j]);
14        }
15        w2 += w3;
16      }
17      lambda[i-imin]= w2;
18    }
19    double y= 0.0;
20    for (i=imin; i<=imax; i++) {
21      y += yd[i]*lambda[i-imin];
22    }
23    return y;
24  }
```

1. L1, 2〜24: メソッド d_lagrange の定義である．引数はメソッド lagrange と同じあり，戻り値は補間で求められた微分値である.
2. L4〜18: 式 (3.14) の Λ' の作成.
3. L19〜22: 微分補間値の計算.
4. L23: 微分補間値を返す.

3.3 ニュートンの補間法

ニュートンの補間法も補間多項式を使う方法の一つである．Lagrange の補間法と同様に，まず，2 次式で補間する場合を検討しよう．2 次式は

$$y = f(x) = ax^2 + bx + c = p_0 + p_1(x - x_0) + p_2(x - x_0)(x - x_1) \tag{3.15}$$

の形に書き換えることができる．この関数は 3 個のパラメータ p_0，p_1，p_2 を使っているから，これらの値を決めるためには 3 組のデータが必要になる．3 組のデータ (x_0, y_0), (x_1, y_1), (x_2, y_2) を補間式 (3.15) に代入すると，

$$\left.\begin{array}{lcl} y_0 &=& p_0 \\ y_1 &=& p_0 + p_1(x_1 - x_0) \\ y_2 &=& p_0 + p_1(x_2 - x_0) + p_2(x_2 - x_0)(x_2 - x_1) \end{array}\right\} \tag{3.16}$$

となる．ここで，未知数であるパラメータ p_0，p_1，p_2 の解は簡単に求めることができて

$$\left.\begin{array}{lcl} p_0 &=& y_0 \\ p_1 &=& (y_1 - p_0)/(x_1 - x_0) \\ p_2 &=& \{(y_2 - p_0)/(x_2 - x_0) - p_1\}/(x_2 - x_1) \end{array}\right\} \tag{3.17}$$

である．これを式 (3.15) に代入すれば 2 次式が決まる．この方法を n 次多項式に拡張すると $p_i\,(0 \le i \le n)$ は

$$\begin{array}{lcl} p_i &=& \{\{\cdots\{\{(y_i - p_0)/(x_i - x_0) - p_1\}/(x_i - x_1) - p_2\}/(x_i - x_2) \\ && - \cdots - p_{i-2}\}/(x_i - x_{i-2}) - p_{i-1}\}/(x_i - x_{i-1}) \end{array} \tag{3.18}$$

である．そこで，式 (3.15) を n 次式での一般型に書き換えると，

$$y = f(x) = \sum_{i=0}^{n} p_i \prod_{j=0}^{i-1}(x - x_j) \tag{3.19}$$

となる．この式 (3.19) をニュートンの補間公式という．プログラムで効率の良い形式に書きなおすと，

$$\begin{array}{lcl} f(x) &=& \{\{\cdots\{\{p_n(x - x_{n-1}) + p_{n-1}\}(x - x_{n-2}) + p_{n-2}\}(x - x_{n-3}) \\ && + \cdots + p_2\}(x - x_1) + p_1\}(x - x_0) + p_0 \end{array} \tag{3.20}$$

となる．図 3.4 は，サイン関数（$\sin x$）のニュートンの補間公式によるグラフである．

表 3.2 は図 3.4 と同様の補間の $x = \pi/8$ における値である．次数が偶数 (データ点が奇数) の場合のみが表に載せてある．次数が 1 増すごとに精度が 1 桁弱良くなっているが，8 次の値は 7 次の値と変わっていない．

式 (3.20) からわかるように，データ点（$x_0, x_1, \cdots, x_n$）は，大きさの順に並んでいる必要はないし，間隔も一様でなくてよい．x の値が等間隔で与えられている場合には，補間計算を能率良く実行するために，いろいろな公式が考案されているが，大差はない．
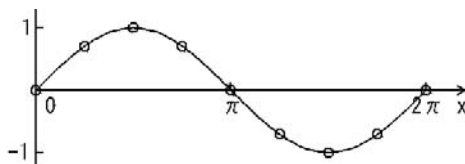
図 3.4　ニュートンの補間法．関数はサイン関数（$\sin x$），データ点の範囲は $[0,\ 2\pi]$，データ点 ∘ 印は等間隔で 9 点とってある．補間には 8 次多項式を使っている．（Z03_04_NewtonInt.java）

表 3.2　ニュートンの補間法におけるデータ点の数と精度の関係．関数はサイン関数 $\sin x$，データ点の範囲は $[0,\ 2\pi]$，データ点は等間隔で 9 点である．補間値を求める点は $x = \pi/8$ で，補間多項式は 1 次から 8 次までである．第 1，3，5 列は補間式の次数であり，第 2，4，6 列は補間値である．（H03_02_NewtonInt.java）

次	補間値	次	補間値	次	補間値
1	0.35355	4	0.38120	7	0.38365
2	0.40533	5	0.37845	8	0.38365
3	0.39460	6	0.38136	真値	0.38268

プログラム (NewtonInt.java) のコンストラクタ (NewtonInt) と，その説明は以下のとおりである．

NewtonInt.java のコンストラクタ (NewtonInt)：

```
 1  public NewtonInt(int n, double[] xd, double[] yd) {
 2    this.xd= xd;
 3    p = new double[n+1];
 4    double w;
 5    for (int i=0; i<=n; i++) {
 6      w = yd[i];
 7      for (int j=0; j<i; j++) {
 8        w = (w-p[j]) / (xd[i]-xd[j]);
 9      }
10      p[i]= w;
11    }
12  }
```

1. L1〜12: コンストラクタ NewtonInt の定義である．引数の n は補間公式の次数であり，データの組数 −1 である．xd はデータの x 座標の配列であり，yd はデータの y 座標の配列である．
2. L2: クラス内の配列 xd に，引数の配列インスタンスの参照を代入する．
3. L3: コンストラクタの外で定義されており，クラス全体が定義域であるフィールド p の配列インスタンスを作る．
4. L5〜11: 多項式のパラメータ p の計算をする．

プログラム (NewtonInt.java) のメソッド (newtonInt) と，その説明は以下のとおりである．

NewtonInt.java のメソッド (newtonInt)：

```
1   public double[] newtonInt(double[] x) {
2     int m= x.length;
3     double[] y= new double[m];
4     double w;
5     for (int k=0; k<m; k++) {
6       w = p[p.length-1];
7       for (int i=p.length-2; i>= 0; i--) {
8         w = w*(x[k]-xd[i]) + p[i];
9       }
10      y[k]= w;
11    }
12    return y;
13  }
```

1. L1〜13: メソッド newtonInt の定義である．引数の x は補間値を求める座標の配列である．戻り値は補間値の配列である．
2. L2: 配列 x の長さを m に代入する．
3. L3: 求める補間値の配列．
4. L5〜11: 求める補間値の数だけループをまわす．
5. L6〜10: 式 (3.20) による補間値の計算．
6. L11: 補間値の配列を返す．

3.4　エルミートの補間法

ラグランジュの補間公式もニュートンの補間公式も $n+1$ 個のデータ点を使って n 次多項式で補間したが，データ点での微分値も与えられていれば，それらも使うことができる．この場合は $2(n+1)$ 個のデータを使うことになるから，近似多項式を $2n+1$ 次式とすることができるので，

$$f_{2n+1}(x) = a_{2n+1}x^{2n+1} + a_{2n}x^{2n} + \cdots a_1x^1 + \cdots a_0 \tag{3.21}$$

である．$2n+2$ 個の係数は $2n+2$ 個の条件

$$\begin{aligned} f_{2n+1}(x_i) &= y_i \quad (i=0,\cdots,n) \\ f'_{2n+1}(x_i) &= y'_i \quad (i=0,\cdots,n) \end{aligned} \tag{3.22}$$

から決めることができる．ラグランジュの補間の場合に補間公式を式 (3.6) の形式にしたように，ラグランジュの補間公式を拡張して

$$f_{2n+1}(x) = \sum_{i=0}^{n} y_i\xi_i(x) + \sum_{i=0}^{n} y'_i\eta_i(x) \tag{3.23}$$

と置くと，$\xi_i(x)$, $\eta_i(x)$ は $(2n+1)$ 次の多項式

$$
\begin{aligned}
\xi_i(x) &= \{1 - 2\Lambda_i'(x_i)(x - x_i)\}\Lambda_i^2(x) \\
\eta_i(x) &= (x - x_i)\Lambda_i^2(x)
\end{aligned}
\tag{3.24}
$$

である．$\Lambda_n(x)$ は式 (3.6) で使われており，

$$
\Lambda_i(x) = \prod_{j=0\,(j\neq i)}^{n} \frac{x - x_j}{x_i - x_j} \tag{3.25}
$$

である．同様に，微分は

$$
\begin{aligned}
f_{2n+1}'(x) &= \sum_{i=0}^{n} y_i \xi_i'(x) + \sum_{i=0}^{n} y_i' \eta_i'(x) \\
\xi_i'(x) &= -2\Lambda_i'(x_i)\Lambda_i^2(x) + 2\{1 - 2\Lambda_i'(x_i)(x - x_i)\}\Lambda_i'(x)\Lambda_i(x) \\
\eta_i'(x) &= \Lambda_i^2(x) + 2(x - x_i)\Lambda_i'(x)\Lambda_i(x)
\end{aligned}
\tag{3.26}
$$

である．ただし，$\Lambda_i'(x)$ は式 (3.14) である．

図 3.5 は，$[-\pi, \pi]$ の領域を 10 等分した 11 の x 点での関数 $(\sin x)$ の値とその微分値をデータとして，エルミートの補間法を使って，その領域のグラフを描いたものである．$\circ$ 印は関数 $(\sin x)$ のデータ値，$\times$ 印はその微分値 $(\cos x)$ である．実線は補間で求めた関数 $(\sin x)$，点線は補間で求めた 1 次導関数 $(\cos x)$ である．

プログラム (Hermit.java) の補間関数値を求めるメソッド (hermit) と，その説明は以下のとおりである．

Hermit.java のメソッド (hermit)：

```
1   public double hermit(double x, double[] xd,  double[] yd, double[] ydd,
2         int min, int max, int n) {
3     int i, j;
4     double[] xi= new double[n+1];
5     double[] eta= new double[n+1];
6
7     if (max-min <= n) {
8       System.out.println("Error: n="+n+" is largeer than max-min.");
9       return 0;
```
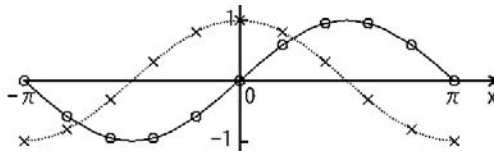
図 3.5　エルミートの補間法．関数 $(\sin x)$ の値とその微分値をデータとし，エルミート補間を使って求めたグラフである．$\circ$ 印は関数 $(\sin x)$ の値，$\times$ 印はその微分値 $(\cos x)$ である．実線は補間で求めた関数 $(\sin x)$，点線は補間で求めた 1 次導関数 $(\cos x)$ である．

```
10 //       System.exit(-1);
11      }
12
13      int imin,imax;
14      int n1= (n+1)/2;
15      int n2= (n+2)/2;
16      double s= xd[max]-xd[min];
17
18      imin= max-n;
19      for (i=min+n2; i<=max-n1; i++) {
20        if ((x-xd[i])*s < 0) {
21          imin= i-n1;
22          break;
23        }
24      }
25      if ((n/2)*2==n && Math.abs(x-xd[i-1])<Math.abs(x-xd[i])) {
26        imin--;
27      }
28      imax= imin+n;
29
30      double w1, w2, w3;
31      for (i=imin; i<=imax; i++) {
32        w1= 1.0;
33        w2= 0.0;
34        for (j=imin; j<=imax; j++) {
35          if (i == j) continue;
36            w3 = 1/(xd[i]-xd[j]);
37            w1 *= (x-xd[j])*w3;
38            w2 += w3;
39        }
40        xi[i-imin]= (1.0-2*w2*(x-xd[i]))*w1*w1;
41        eta[i-imin]= (x-xd[i])*w1*w1;
42      }
43
44      double y= 0.0;
45      for (i=imin; i<=imax; i++) {
46        y += yd[i]*xi[i-imin]+ydd[i]*eta[i-imin];
47      }
48      return y;
49    }
```

1. L1, 2〜49: メソッド hermit の定義である．引数の x は補間値を求める点の x 座標，xd, yd, ydd は既知のデータ点 (x, y) と微分値の配列，min は使用する配列の最小添字番号，max は使用する配列の最大添字番号であり，n は補間式の次数である．戻り値は補間値である．
2. L4: 式 (3.23) で関数値 yd[i] との積和を作る項の式 $\xi(x)$ の配列．
3. L5: 式 (3.23) で微分値 ydd[i] との積和を作る項の式 $\eta(x)$ の配列．
4. L16: 変数 s に xd[max]−xd[min] を代入し，昇順なら正，降順なら負とする．x 座標は昇順でも降順でもかまわない．

5. L18〜27: 内挿の場合には，使うデータ点の中央近くに x が来るようにデータ点を選び，外挿の場合には x に近い点から選んで，補間に使う最小添字番号を決定する．
6. L28: 補間に使う最大添字番号を決定する．
7. L30〜42: 式 (3.24) の項 $\xi(x)$，$\eta(x)$ の作成.
8. L44〜47: 補間値の計算.
9. L48: 補間値を返す.

プログラム (Hermit.java) の補間微分値を求めるメソッド (d_hermit) の一部と，その説明は以下のとおりである.

Hermit.java のメソッド (d_hermit) の一部：

```
 1  public double d_hermit(double x, double[] xd,  double[] yd, double[] ydd,
 2        int min, int max, int n) {
 3  中略
 4    double w1, w2, w3, w4, w5=0.0;
 5    for (i=imin; i<=imax; i++) {
 6      w1= 1.0;
 7      w2= 0.0;
 8      w4= 0.0;
 9      for (k=imin; k<=imax; k++) {
10        if (k == i) continue;
11        w3 = 1/(xd[i]-xd[k]);
12        w1 *= (x-xd[k])*w3;
13        w2 += w3;
14        w5= 1.0/(xd[i]-xd[k]);
15        for (j=imin; j<=imax; j++) {
16          if (j==i || j==k) continue;
17          w5 *= (x-xd[j])/(xd[i]-xd[j]);
18        }
19        w4 += w5;
20      }
21      xi_d[i-imin]= -2*w2*w1*w1 + 2*(1.0-2*w2*(x-xd[i]))*w4*w1;
22      eta_d[i-imin]= w1*w1 + 2*(x-xd[i])*w4*w1;
23    }
24    double y= 0.0;
25    for (i=imin; i<=imax; i++) {
26      y += yd[i]*xi_d[i-imin]+ydd[i]*eta_d[i-imin];
27    }
28    return y;
29  }
```

1. L1, 2〜29: メソッド d_hermit の定義である．引数はメソッド hermit と同じであり，戻り値は微分値 (1 次導関数) である．
2. L4〜23: 式 (3.26) の項 $\xi'(x)$，$\eta'(x)$ の作成.
3. L24〜27: 式 (3.26) で微分値の補間計算をする.
4. L28: 補間微分値を返す.

3.5　スプライン補間

補間式としてスプライン関数 $y = S(x)$ を使う場合をスプライン補間という．スプライン関数については，第 12 章に解説があるが，ここでは，補間に使う B スプライン補間と 3 次スプライン補間について述べる．一般的にスプライン関数は区分的多項式であり，与えられたデータ点すべてにわたって単一の多項式を使うこれまでの補間とは異なり，データ点を区分して局所的な多項式を作り，それらを使うものである．n 次の B スプライン補間の場合には，$n-1$ 次の微分まで連続である．x 軸上で区分する点を節点といい，データ点 x_i を節点として使う場合もあるが，データ点とはあまり関係がない適当な点をとる場合もある．その場合には，後で示す条件を満たす必要がある．節点は ξ_i で表すことにする．

3.5.1　B スプラインによる補間

B スプライン (Basis spline) とは局所的な多項式 $N_{m,j}(x)$ として作られ，スプライン関数がそれらを基底関数とする線形結合

$$S(x) = \sum c_j N_{m,j}(x) \tag{3.27}$$

で定義されるものである．この場合には，データ点を節点にしない．多項式の次数 i に応じて，i 次の B スプライン，あるいは $i+1$ 階の B スプラインといわれる．3 次 (4 階) の B スプラインがよく使われるので，説明は主として 3 次の B スプラインで行うことにする．

B スプラインにはいくつかの作り方があるが，補間の場合にはドュブア・コックスのアルゴリズムによるものが使われる．この方法によると m 階の B スプライン $M_{m,j}(x)$ の値は次の漸化式で求められ，1 階の関数から順次高階の関数が作られる．

$$\begin{aligned} M_{1,j}(x) &= \begin{cases} \dfrac{1}{\xi_j - \xi_{j-1}} & (\xi_{j-1} \le x < \xi_j) \\ 0 & (\text{その他}) \end{cases} \\ M_{m,j}(x) &= \frac{(x-\xi_{j-m})M_{m-1,j-1}(x) + (\xi_j - x)M_{m-1,j}(x)}{\xi_j - \xi_{j-m}} \quad (m = 2, \cdots) \end{aligned} \tag{3.28}$$

図 3.6 はドュブア・コックスのアルゴリズムによる B スプラインであり，上から m 番目が m 階の関数である．一番上の 1 階の関数は $1 \le j \le 8$ の 8 個示されており，それぞれが 2 個の節点を使っている．図の ∘ 印は，各階の一番右側 ($j = 8$) の B スプラインに使われている節点を示している．関数の数は，階が上がるたびに 1 個ずつ減っており，4 階では $4 \le j \le 8$ の 5 個である．逆に，1 個の関数が使う節点の数は増加し，4 階では 5 点を使っている．B スプラインに付けられる番号の j は，その関数に使われている一番右側の節点の番号となる．この B スプライン $M_{m,j}(x)$ は規格化
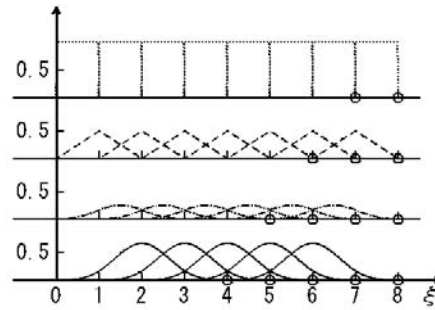
図 3.6　ドュブア・コックスのアルゴリズムによる B スプライン．上から m 番目が m 階である．一番下の 4 階だけは，規格化された B スプライン $N_{m,j}(x)$ であり，4 倍の大きさになっている．$\circ$ 印は，各階の一番右側 ($j=8$) の B スプラインに使われている節点である．(Z03_06_deBoor_Cox_exp.java)

されておらず，不便なこともあるので，

$$N_{m,j}(x) = (\xi_j - \xi_{j-m}) M_{m,j}(x) \tag{3.29}$$

で与えられる，規格化された B スプラインを使うことが多い．

m 階の B スプライン $N_{m,j}(x)$ を基底関数とするスプライン補間式は

$$S(x) = \sum_{j=1}^{\nu+m} c_j N_{m,j}(x) \tag{3.30}$$

である．補間式が定義される領域は，節点 ξ_1 から ξ_ν を覆う領域であり，そのために，必要とされる B スプラインは $j = 1, \cdots, \nu+m$ である．しかし，ある一つの点 x で実際に寄与する B スプラインの数は，その点でゼロでない関数だけであるから m 個である．例えば，図 3.6 の一番下の 4 階の B スプラインでゼロでないものは，区間 ξ_3 〜ξ_5 において 4 個であることがわかる．しかし，データ点が節点に重なっている場合には，ゼロでないものは 3 個になってしまう．また，一番左端の $j=1$ の関数は節点 ξ_{1-m} から ξ_1 を使うので，全体としては $j = 1-m, \cdots, \nu+m$ で，$\nu+2m$ 個の節点が必要となる．

まとめると，ξ_1 から ξ_ν までの領域を覆う m 階のスプライン関数を作る場合には，両端の付加節点 m 個ずつを加えて，節点は $\nu+2m$ 個が必要であり，そのときできる B スプラインは $\nu+m$ 個である．そこで，スプライン関数の係数を決めるためには，必要なデータ点の数も $\nu+m$ 個である．

プログラム (Bspl.java) のメソッド (bspl) と，その説明は以下のとおりである．

Bspl.java のメソッド (bspl)：

```
1  public int bspl(double xp, double[] xi, double[] N) {
2    int nxi= xi.length;
3    int ior= N.length;
```

```
 4     int jxi;
 5     int iorm1= ior-1;
 6     int iorp1= ior+1;
 7     int nximior= nxi-ior;
 8     double[][] M= new double[iorm1][iorp1];
 9     int i, j, k, l;
10
11     for (j=0; j<nxi; j++) {
12       if (xp < xi[j]) break;
13     }
14     if (j>nximior) {
15       jxi= nximior;
16     } else {
17       jxi=j;
18     }
19
20     for (i=0; i<iorm1; i++) {
21       for (j=0; j<iorp1; j++) {
22         M[i][j]= 0.0;
23       }
24     }
25     M[0][1]= 1.0/(xi[jxi]-xi[jxi-1]);
26     for (i=1; i<iorm1; i++) {
27       for (j=1; j<=i+1; j++) {
28         k= jxi+j-i-2;
29         l= jxi+j-1;
30         M[i][j]=((xp-xi[k])*M[i-1][j-1] + (xi[l]-xp)*M[i-1][j])
31           /(xi[l]-xi[k]);
32       }
33     }
34     for (j=1; j<iorp1; j++) {
35       k= jxi+j-ior-1;
36       l= jxi+j-1;
37       N[j-1]=(xp-xi[k])*M[iorm1-1][j-1]+(xi[l]-xp)*M[iorm1-1][j];
38     }
39   中略
40     return jxi;
41   }
```

1. L1〜41: メソッド bspl の定義である．引数 xp は B スプライン値を求める x 座標，xi は指定した節点の配列，N は戻される B スプラインの値の配列で，要素数はゼロでない B スプラインの数 (＝階数) だけである．戻り値は，ゼロでない最初の B スプラインの番号である．
2. L2, 3: 節点の数と階数を nxi, ior に代入する．
3. L5, 6: 式 (3.28) での M の計算のため，階数 -1 と階数 $+1$ を定義する．
4. L8: 式 (3.28) での M の定義は，階数 -1 までとし，最後の規格化された値は引数の N に代入する．
5. L11〜13: 座標点 xp が節点のどこにあるかを調べ，j に代入する．
6. L14〜18: xp が最後の B スプラインの左端を超えていたら，最後の B スプライ

ンの番号を jxi に指定し，超えていなかったら j を jxi に指定する．

7. L20〜24: M をゼロで初期化する．
8. L25: M のうち，階数が 1 で 2 番目の M[0][1] のみ，式 (3.28) の第 1 式の値を代入する．
9. L26〜33: 式 (3.28) の第 2 式で 2 から (階数 −1) まで繰り返すループである．
10. L27〜32: 式 (3.28) の第 2 式のループを繰り返す．
11. L34〜38: 規格化された値を N に代入する．
12. L40: ゼロでない最初の B スプラインの位置を戻す．

式 (3.30) の係数 c_j を決定するためには，データ点は $\nu + m$ 個必要であり，それらの値を代入して，c_j に関する連立方程式

$$\sum_{j=1}^{\nu+m} c_j N_{m,\,j}(x_i) = y_i \quad (i = 1, \cdots, n) \tag{3.31}$$

を作り，その解を求めればよい．この連立方程式を行列表現すると

$$N\mathbf{c} = \mathbf{y} \tag{3.32}$$

である．1 個の B スプライン $N_{m,\,j}(x)$ がゼロでない範囲は，節点 $\xi_{j-m} \sim \xi_j$ の m 区間であるから，この係数行列 N の各行の要素でゼロでないものは高々 m 個である．節点がデータ点と重なっていると，その節点を左右どちらかの端の節点とする B スプラインの値はゼロになるから，ゼロでない要素の数は $m-1$ 個となり，さらに節点が重複点の場合には 1 個の場合もある．次の「B スプラインによる滑らかな関数の補間」の例では，3 次のスプライン関数で，11 個のデータと B スプラインを使っている．この場合の係数行列 N は

$$N = \begin{pmatrix}
\circ & & & & & & & & & & \\
\circ & \circ & \circ & \circ & & & & & & & \\
 & \circ & \circ & \circ & & & & & & & \\
 & & \circ & \circ & \circ & & & & & & \\
 & & & \circ & \circ & \circ & & & & & \\
 & & & & \circ & \circ & \circ & & & & \\
 & & & & & \circ & \circ & \circ & & & \\
 & & & & & & \circ & \circ & \circ & & \\
 & & & & & & & \circ & \circ & \circ & \\
 & & & & & & & \circ & \circ & \circ & \circ \\
 & & & & & & & & & & \circ
\end{pmatrix} \tag{3.33}$$

である．$\circ$ 印がゼロでない要素であり，他はすべてゼロである．節点の数は 15 個であり，両端は 4 重点となっているので，最初の第 1 行と最後の第 11 行にはゼロでない要素が 1 個ずつしかない．途中の第 3 行から第 9 行までは 3 個であるが，3 番目から 9 番目ではデータ点と節点が重なっているからである．重なっていない第 2 行と第 10 行は階数と同じ 4 個がゼロでない．また，このような行列 N の構成から，スプライン関数が局所的であるということがわかる．すなわち，ラグランジュの公式 (3.6) においては，一つの y_i の値に変化があった場合に，補間式全体に影響が及ぶが，式 (3.32)

においては，一つの y_i の値に変化があった場合に，直接影響を受ける解の c_j は高々 4 個であり，他の解の受ける影響は間接的になるからである．

B スプラインによる滑らかな関数の補間

図 3.7 はドュブア・コックスのアルゴリズムによる 4 階 ($m = 4$) の B スプライン補間の例である．データ点の数が 11 点であるから，式 (3.30) の和の項は 11 項であり，11 個の B スプラインが必要となる．そこで，4 階の場合には必要とする節点の数は 15 個となる．最も左側の B スプラインでは，最も右側の節点だけが補間領域に入り，他の 4 点は左側に出ていなければならない．このことは，右側についても同じであり，結局，補間領域内部に収まる節点の数は 7 点となる．この例では，補間領域外部の付加した節点は，左右でそれぞれ 4 重点としてある．節点の数ととり方には制限があり，データ点の個数 n，内部節点の個数 ν，B スプラインの階数 m の間には関係 $n = \nu + m$ がなければならない．また，ホイットニ (Whitney) の条件

$$\begin{cases} x_1 < \xi_1 < x_{1+m} \\ \qquad \cdots \\ x_{n-m} < \xi_\nu < x_n \end{cases} \tag{3.34}$$

を満足しないと，連立方程式の係数行列 N にすべての要素がゼロである列ができてしまい，連立方程式が解けなくなってしまう．

図 3.7 での関数は $f(x) = \dfrac{1}{1 + (x-3)^2}$ であり，変動のわりにデータ点の数が粗い原点の近くではあまり近似が良くない．

プログラム (Bspline_Interpolation.java) は，ユーザが書くプログラムから呼ばれるメソッド (bspline_Interpolation) であり，その中で同じクラスにある 3 個のメソッドを呼ぶようになっている．メソッド (bspline_Interpolation) と，その説明は以下のとおりである．
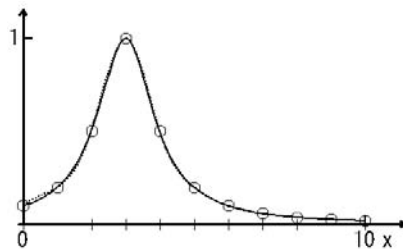
図 3.7　3 次の B スプラインによる補間．関数は $f(x) = \dfrac{1}{1 + (x-3)^2}$ である．データ点は ∘ 印の 11 点 (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) であり，補間領域は [0, 10] である．補間領域内の節点は 7 点 (2, 3, 4, 5, 6, 7, 8) であり，領域の両側に付け加えた付加節点 (0, 0, 0, 0) と (10, 10, 10, 10) を合わせて 15 点で，x 軸上に短いバーで示されている．点線は補間曲線であり，実線は真の曲線である．(Z03_07_Bspline_Interpolation)

Bspline_Interpolation.java のメソッド (bspline_Interpolation)：

```
 1  public double bspline_Interpolation(double[] x, double[] y,
 2      double[] xi, double[] xd, double[] yd, int ior) {
 3  中略
 4    int nxd= xd.length;
 5    double[] c= new double[nxd];
 6    double[][] a= new double[nxd][nxd];
 7    keisuu_Gyouretsu(xd, xi, a, ior);
 8    Gauss_Shoukyohou gs= new Gauss_Shoukyohou();
 9    c= gs.gauss_Shoukyohou(a,ydw);
10    return hokanchi_keisan(x, y, xi, c, ior);
11  }
```

1. L1, 2〜11: メソッド bspline_Interpolation の定義である．引数の x は補間値を求める x 座標の配列，y は補間値が戻される配列，xi は指定した節点の配列，xd はデータの x 座標の配列，yd はデータの y 座標の配列，ior は B スプラインの階数である．戻り値は補間曲線の積分値である．
2. L5: c は補間式の B スプラインの係数である．
3. L6: a は c を求めるための連立方程式の係数行列である．
4. L7: メソッド keisuu_Gyouretsu を呼んで，連立方程式の係数行列を作成する．
5. L8, 9: クラス Gauss_Shoukyohou のメソッド gauss_Shoukyohou を呼んで，連立方程式の解 c を求める．gauss_Shoukyohou は引数の配列の値を変更してしまうので，配列 yd の値を作業用の配列 ydw に前もってコピーしておき，それを引数として使っている．
6. L10: メソッド hokanchi_keisan を呼んで，指定した配列 x の座標値に対する補間値を配列 y に求める．その際，補間曲線の積分値が戻り値として戻される．

メソッド (keisuu_Gyouretsu) とその説明は以下のとおりである．

Bspline_Interpolation.java のメソッド (keisuu_Gyouretsu)：

```
 1  public void keisuu_Gyouretsu(double[] xi, double[] xd,
 2      double[][] a, int ior) {
 3    int nxi= xi.length;
 4    int nxd= xd.length;
 5    double[] rn= new double[ior];
 6    int ixi, ixmo;
 7    //  データ点と節点のホイットニ条件の検証
 8    for (int i=ior; i<nxd; i++) {
 9      if (xd[i-ior] < xi[i] && xi[i] < xd[i]) continue;
10      System.out.println("Whitney condition is violated");
11      System.exit(1);
12    }
13    for (int i=0; i<nxd; i++) {  //  配列 a の初期化
14      for (int j=0; j<nxd; j++) {
15        a[i][j]=0.0;
16      }
17    }
```

```
18      Bspl bs= new Bspl();
19      for (int i=0; i<nxd; i++) {
20        ixi= bs.bspl(xd[i], xi, rn);
21        ixmo= ixi-ior;
22        for (int k=0; k<ior; k++) {
23          a[i][ixmo+k]=rn[k];
24         }
25       }
26      return;
27    }
```

1. L1, 2〜27: メソッド keisuu_Gyouretsu の定義である．引数の xi は指定した節点の配列，xd はデータの x 座標の配列，a は係数行列の 2 次元配列，ior は B スプラインの階数である．
2. L5: データ点における B スプライン値が戻される作業用配列．
3. L8〜12: データ点と節点のホイットニ条件の検証をする．違反していたら，警告を出し処理を中断する．
4. L13〜17: 配列 a をゼロに初期化する．
5. L18: B スプライン値を戻すクラスのインスタンスを作成する．
6. L19〜25: データ点の数だけループをまわす．
7. L20: B スプライン値を求めるメソッドを呼び，配列 rn に B スプライン値を，また，戻り値としてゼロでない最初の B スプラインの番号を ixi に得る．
8. L22〜24: 配列 a のしかるべき要素に，ゼロでない B スプライン値を代入する．

メソッド (hakidashihou) は第 7 章で説明する．メソッド (hokanchi_keisan) とその説明は以下のとおりである．

Bspline_Interpolation.java のメソッド (hokanchi_keisan)：

```
 1   public double hokanchi_keisan(double[] x, double[] y, double[] xi,
 2          double[] c, int ior) {
 3     int nx= x.length;
 4     int nxi= xi.length;
 5     int nxd= c.length;
 6     double[] rn= new double[ior];
 7     int ixi, iximor;
 8     Bspl bs= new Bspl();
 9     for (int ix=0; ix<nx; ix++) {
10       ixi= bs.bspl(x[ix], xi, rn);
11       iximor= ixi-ior;
12       y[ix]=0.0;  //  係数と B スプラインの積和
13       for (int i=0; i<ior; i++) {
14         y[ix]=y[ix]+c[iximor+i]*rn[i];
15       }
16      }
17     double rinteg=0.0;  //  積分値の計算
18     for (int i=0; i<nxd; i++) {
19       rinteg= rinteg+(xi[i+ior]-xi[i])*c[i];
```

```
20          }
21        return rinteg/ior;
22    }
```

1. L1, 2〜22: メソッド hokanchi_keisan の定義である．引数 x は補間値を求める x 座標の配列，y は補間値が戻される配列，xi は指定した節点の配列，c は B スプラインにかかる係数，ior は B スプラインの階数である．戻り値は補間関数の積分値である．
2. L6: 補間値を求める点における B スプライン値が戻される作業用配列．
3. L8: B スプライン値を戻すクラスのインスタンスを作成する．
4. L9〜16: 補間値を求める点の数だけ繰り返す．
5. L10: B スプライン値を戻すメソッドを呼び，配列 rn に B スプライン値を，戻り値としてゼロでない最初の B スプラインの番号を ixi に得る．
6. L12〜15: 補間値を計算し，配列 y に代入する．
7. L17〜21: 補間関数の積分値を計算し，戻り値とする．

拡張 B スプラインによる不連続のある関数の補間

図 3.8 は，$x = 4$ に微分値の不連続が，$x = 10$ に関数値の不連続がある関数の場合である．不連続がある場合の B スプラインを拡張 B スプラインということがある．関数は

$$f(x) = \begin{cases} \dfrac{1}{2+\frac{3}{4}(x-2)(x-2)} & (x < 4) \\ \dfrac{1}{1+(x-6)(x-6)} & (4 \leq x < 10) \\ \dfrac{1}{2+(x-12)(x-12)} + 0.3 & (10 \leq x) \end{cases} \tag{3.35}$$

である．データ点と節点は表 3.3 に与えられている．データ点は 25 個あり，節点は内部節点 21 個と付加節点 8 個がある．$x = 4$ に微分値の不連続があるので節点を 3 重とし，$x = 10$ には関数値の不連続があるので節点を 4 重としている．
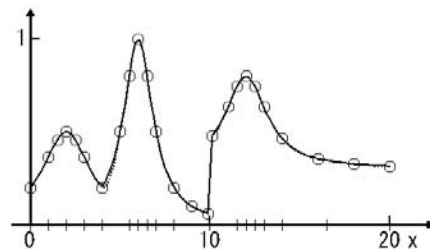
図 3.8　3 次の拡張 B スプラインによる不連続な関数の補間．関数は式 (3.35) に与えられている．データ点 ∘ 印と x 軸上に短いバーで示されている節点は表 3.3 に与えられている．点線は補間曲線であり，実線は真の曲線である．(Z03_08_Bspline_Extended)

表 3.3　拡張 B スプラインによる不連続な関数の補間用データ点と節点．データ点は 25 個あり，節点は内部節点 21 個と付加節点 8 個がある．

データ点	0, 1, 1.5, 2, 2.5, 3, 4, 5, 5.5, 6, 6.5, 7, 8, 9, 9.9, 10.1, 11, 11.5, 12, 12.5, 13, 14, 16, 18, 20
節点	0, 0, 0, 0, 1, 2, 3, 4, 4, 4, 5.5, 6, 6.5, 7, 8, 10, 10, 10, 10, 11.5, 12, 12.5, 13, 14, 16.5, 20, 20, 20, 20

B スプラインによる平面曲線図形の補間

図 3.9 は，♡ 型を作るための 23 点のデータ点 (∘ 印) を指定し，B スプラインによる平面曲線図形の補間を行って，♡ 型を描画したものである．

データ点は x と y の関係 (x_i, y_i) であるが，媒介変数 t を導入して，$x = x(t)$，$y = y(t)$ の二つの補間関数を考慮する．変数 t の座標値は，点 (x_0, y_0) から曲線に沿って計る距離を目安として，$t_j = \sum_{i=1}^{j} \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}$ で定義する．表 3.4 は ♡ 型を作る 23 のデータ点である．

表 3.5 は ♡ 型を作る 19 の節点を $x = x(t)$，$y = y(t)$ の二つの関数に対して別々に作った (ξ_j, η_j) である．

$x = x(t)$ は $t = 4.5$ で滑らかな関数であるが，$y = y(t)$ は $t = 4.5$ で尖った値をもっている．そこで，$x = x(t)$ に関する節点は $t = 4.5$ において重点とはしていないが，$y = y(t)$ に関する節点は $t = 4.5$ において 3 重点としている．t を区間 $[0, 9]$ で 100 分割して，$x = x(t)$ と $y = y(t)$ について，それぞれ独立に補間法を適用し，同じ t の値の x, y を座標として図 3.9 は描かれている．補間法を適用したこの図では，あまり滑らかでない部分もあるが，後で述べる B スプラインを使った図形作成法ではもっと滑らかな ♡ 型が描かれている．
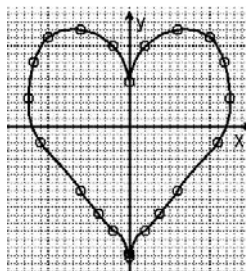
図 3.9　B スプライン補間による ♡ 型の描画．23 点のデータ点を指定し，B スプラインによる平面曲線図形の補間を行った．∘ 印はデータ点である．媒介変数 t を導入して，19 の節点を $x = x(t)$，$y = y(t)$ の二つの関数に対して別々にとった．(Z03_09_Bspline_Heart.java)

表 3.4　♡ 型を作る 23 のデータ点．x と y は座標点であり，t は媒介変数である．

x	0.00	−0.20	−0.60	−1.00	−1.18	−1.25	−1.10	−0.60
y	0.55	1.00	1.20	1.10	0.80	0.35	−0.20	−0.80
t	0.00	0.49	0.93	1.35	1.70	2.15	2.72	3.50
x	−0.38	−0.20	−0.01	0.00	0.01	0.20	0.38	0.60
y	−1.08	−1.30	−1.58	−1.62	−1.58	−1.30	−1.08	−0.80
t	3.86	4.14	4.48	4.50	4.52	4.86	5.14	5.50
x	1.10	1.25	1.18	1.00	0.60	0.20	0.00	
y	−0.20	0.35	0.80	1.10	1.20	1.00	0.55	
t	6.28	6.85	7.30	7.65	8.07	8.51	9.00	

表 3.5　♡ 型を作る 19 の節点．

ξ	0.6	1.4	2.0	2.6	3.3	3.6	3.9	4.2	4.47	4.5
η	0.6	1.2	1.8	2.4	3.2	3.8	4.2	4.3	4.5	4.5
ξ	4.53	4.8	5.1	5.4	5.7	6.4	7.0	7.6	8.4	
η	4.5	4.7	4.8	5.2	5.8	6.6	7.2	7.8	8.4	

3.5.2　3 次スプライン補間

ここでは，3 次関数がもつ特徴を使ったもう一つのスプライン補間式を作る．これは，3 次の B スプラインとは異なるスプライン関数である．この場合には，節点をデータ点と同じにとっておく必要があり，それらを

$$x_0 < x_1 < \cdots < x_{n-1} \tag{3.36}$$

とする．3 次のスプライン関数 $S(x)$ として，区間 $[x_0, x_{n-1}]$ で 2 次導関数まで連続であり，しかも，

$$S(x_i) = y_i \quad (i = 0, \cdots, n-1) \tag{3.37}$$

を満たすものを作る．3 次式であるから，

$$S(x) = ax^3 + bx^2 + cx + d \tag{3.38}$$

と表せる．まだ未知であるが，各節点での 1 次微係数 $S'(x_i)$ を M_i と置くと，$x = x_{i-1}$，x_i において，

$$\begin{aligned} S(x_i) &= ax_i^3 + bx_i^2 + cx_i + d = y_i \\ S(x_{i-1}) &= ax_{i-1}^3 + bx_{i-1}^2 + cx_{i-1} + d = y_{i-1} \\ S'(x_i) &= \frac{1}{3}ax_i^2 + \frac{1}{2}bx_i + c = M_i \\ S'(x_{i-1}) &= \frac{1}{3}ax_{i-1}^2 + \frac{1}{2}bx_{i-1} + c = M_{i-1} \end{aligned} \tag{3.39}$$

となる．この連立方程式を解いて式 (3.38) に代入して整理すると，区間 $[x_{i-1}, x_i]$ において，

$$\begin{aligned} S(x) &= M_{i-1}\frac{(x_i-x)^2(x-x_{i-1})}{h_i^2} - M_i\frac{(x_i-x)(x-x_{i-1})^2}{h_i^2} \\ &\quad + y_{i-1}\frac{(x_i-x)^2\{2(x-x_{i-1})+h_i\}}{h_i^3} \\ &\quad + y_i\frac{(x-x_{i-1})^2\{2(x_i-x)+h_i\}}{h_i^3} \end{aligned} \tag{3.40}$$

となる．ただし，$h_i = x_i - x_{i-1}$ としている．その 2 次導関数は，

$$\begin{aligned} S''(x) &= -2M_{i-1}\frac{2x_i+x_{i-1}-3x}{h_i^2} - 2M_i\frac{2x_{i-1}+x_i-3x}{h_i^2} \\ &\quad +6\frac{y_i-y_{i-1}}{h_i^3}(x_i+x_{i-1}-2x) \end{aligned} \tag{3.41}$$

である．ここで，2 次導関数が各節点で連続であるという条件を課すると，点 x_i では，$S''(x_i-) = S''(x_i+)$ より

$$\begin{aligned} &\frac{2M_{i-1}}{h_i} + \frac{4M_i}{h_i} - 6\frac{y_i-y_{i-1}}{h_i^2} \\ &\quad = -\frac{4M_i}{h_{i+1}} - \frac{2M_{i+1}}{h_{i+1}} + 6\frac{y_{i+1}-y_i}{h_{i+1}^2} \end{aligned} \tag{3.42}$$

と置けるので，

$$\frac{1}{h_i}M_{i-1} + 2\left(\frac{1}{h_i}+\frac{1}{h_{i+1}}\right)M_i + \frac{1}{h_{i+1}}M_{i+1} = 3\frac{y_i-y_{i-1}}{h_i^2} + 3\frac{y_{i+1}-y_i}{h_{i+1}^2} \tag{3.43}$$

となる．次に，

$$\lambda_i = \frac{h_{i+1}}{h_i+h_{i+1}}, \quad \mu_i = 1-\lambda_i \tag{3.44}$$

と置くと式 (3.43) は最終的に

$$\begin{aligned} \lambda_i M_{i-1} + 2M_i + \mu_i M_{i+1} &= 3\lambda_i\frac{y_i-y_{i-1}}{h_i} + 3\mu_i\frac{y_{i+1}-y_i}{h_{i+1}} = b_i \\ &\quad \text{ただし，} i = 1, \cdots, n-2 \end{aligned} \tag{3.45}$$

となる．n 個の未知数 M_i に対して $n-2$ の方程式を立てたことになる．残る 2 個の方程式を，$i = 0,\ n-1$，すなわち両端での端条件を

$$\begin{aligned} 2M_0 + \mu_0 M_1 &= b_0 \\ \lambda_{n-1}M_{n-2} + 2M_{n-1} &= b_{n-1} \end{aligned} \tag{3.46}$$

とすることによって，式 (3.45) と式 (3.46) はまとめられて，連立方程式

$$A\mathbf{M} = \mathbf{b} \tag{3.47}$$

となる．ここで，$A, \mathbf{M}, \mathbf{b}$ はそれぞれ

$$A = \begin{pmatrix} 2 & \mu_0 & & \cdots & & & 0 \\ \lambda_1 & 2 & \mu_1 & \cdots & & & \\ & \lambda_2 & 2 & \cdots & & & \\ & & & \cdots & & & \\ & & & \cdots & 2 & \mu_{n-2} & \\ & & & \cdots & \lambda_{n-2} & 2 & \mu_{n-2} \\ 0 & & & \cdots & & \lambda_{n-1} & 2 \end{pmatrix}, \mathbf{M} = \begin{pmatrix} M_0 \\ M_1 \\ \cdot \\ M_{n-2} \\ M_{n-1} \end{pmatrix}, \mathbf{b} = \begin{pmatrix} b_0 \\ b_1 \\ \cdot \\ b_{n-2} \\ b_{n-1} \end{pmatrix} \tag{3.48}$$

であり，行列 A は 3 重対角行列となる．

一方，スプライン関数が周期関数であり，その周期が (x_0, x_{n-1}) である場合には，$x_{n-1} = x_0$ と置けるから，未知数と方程式の数が 1 個ずつ減少して，式 (3.48) は

$$A = \begin{pmatrix} 2 & \mu_1 & & \cdots & & & \lambda_1 \\ \lambda_2 & 2 & \mu_2 & \cdots & & & \\ & \lambda_3 & 2 & \cdots & & & \\ & & & \cdots & & & \\ & & & \cdots & 2 & \mu_{n-3} & \\ & & & \cdots & \lambda_{n-2} & 2 & \mu_{n-2} \\ \mu_{n-1} & & & \cdots & & \lambda_{n-1} & 2 \end{pmatrix}, \mathbf{M} = \begin{pmatrix} M_1 \\ M_2 \\ \cdot \\ M_{n-2} \\ M_{n-1} \end{pmatrix}, \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \cdot \\ b_{n-2} \\ b_{n-1} \end{pmatrix} \tag{3.49}$$

となる．行列 A は 3 重対角のほかに，右上と左下の要素がゼロでない．

プログラム (Spline3_IntNP.java) のメソッド (spline3_IntNP) と，その説明は以下のとおりである．連立方程式の詳しい解法は，後の第 7 章に述べられている．

Spline3_IntNP.java のメソッド (spline3_IntNP)：

```
 1    public void spline3_IntNP(double[] x, double[] y,
 2      double b0, double bnm1, double mu0, double lamnm1,
 3      double[] xd, double[] yd, double[] M) {
 4    int nxd= xd.length;
 5    int nx= x.length;
 6    double[] p= new double[nxd];
 7    double[] q= new double[nxd];
 8    double[] u= new double[nxd];
 9    double[] b= new double[nxd];
10    double[] mu= new double[nxd];
11    double[] lam= new double[nxd];
12    double[] h= new double[nxd];
13    double[] M= new double[nxd];
14    double xjm1, xj, xi, hj1,hj2,hj3;
15    int i, k, jm1, j;
16    b[0]= b0;
17    b[nxd-1]= bnm1;
18    mu[0]= mu0;
```

```
19      lam[nxd-1]= lamnm1;
20      h[1]= xd[1]-xd[0];
21      for (int id=1; id<nxd-1; id++) {
22        h[id+1]= xd[id+1]-xd[id];
23        lam[id]= h[id+1]/(h[id]+h[id+1]);
24        mu[id]= 1.0-lam[id];
25        b[id]= 3.0*(lam[id]*(yd[id]-yd[id-1])/h[id]
26              +mu[id]*(yd[id+1]-yd[id])/h[id+1]);
27      }
28      p[0]= 2.0;
29      q[0]=-mu[0]/p[0];
30      u[0]= b[0]/p[0];
31      for (int id=1; id<nxd; id++) {
32        p[id]= lam[id]*q[id-1]+2.0;
33        q[id]=-mu[id]/p[id];
34        u[id]= (b[id]-lam[id]*u[id-1])/p[id];
35      }
36      M[nxd-1]= u[nxd-1];
37      for (int id=nxd-2; id>=0; id--) {
38        M[id]= q[id]*M[id+1]+u[id];
39      }
40      for (i=0; i<nx; i++) {
41        xi= x[i];
42        for (k=1; k<nxd-1; k++) {
43          if (xi <= xd[k]) break;
44        }
45        j=k;
46        jm1= j-1;
47        xj= xd[j]-xi;
48        xjm1= xi-xd[jm1];
49        hj1= h[j];
50        hj2= hj1*hj1;
51        hj3= hj2*hj1;
52        y[i]= M[jm1]*xj*xj*xjm1/hj2
53           -M[j]*xjm1*xjm1*xj/hj2
54           +yd[jm1]*xj*xj*(2.0*xjm1+hj1)/hj3
55           +yd[j]*xjm1*xjm1*(2.0*xj+hj1)/hj3;
56      }
57    }
```

1. L1〜3〜57: メソッド spline3_IntNP の定義である．引数 x は補間値を求める x 座標の配列，y は補間値が戻される配列である．b0 は端条件式 (3.46) の第 1 式の右辺定数 (b_0)，bnm1 は第 2 式の右辺定数 (b_{n-1})，mu0 は第 1 式の第 2 係数 (μ_0)，lamnm1 は第 2 式の第 1 係数 (λ_{n-1}) である．xd はデータの x 座標の配列，yd はデータの y 座標の配列である．
2. L4: 変数 nxd にデータ点の数を代入．
3. L5: 変数 nx に補間値を求める x 点の数を代入．
4. L16〜27: 係数行列の μ と λ，および右辺のベクトル $\mathbf{b}$ の要素を作る．対角要素の 2.0 は L28 と L32 で使われている．

5. L28〜35: 前進消去を行う．
6. L36〜39: 後退代入で解 M を求める．
7. L40〜56: 補間値を求める点の数だけループをまわす．
8. L42〜44: 補間値を求める点 x の位置がどの節点 (データ点と同一) のところにあるかを調べる．
9. L52〜55: 式 (3.40) の計算をして，補間値を求める．

以上の手続きで連立方程式を解き，式 (3.47) の M_i を求めれば，式 (3.40) の 3 次のスプライン関数が決定される．図 3.10 は 3 次の非周期スプラインによる補間である．関数は図 3.7 と同じ $f(x) = \dfrac{1}{1+(x-3)^2}$ である．3 次の B スプラインを使った補間の図 3.7 と比較すると，この 3 次のスプライン関数によるほうが良いようである．

図 3.11 は 3 次の周期スプラインによる平面曲線の補間である．データ点と節点は同一であり，図には ∘ 印で示されている．データは始点 (1.3, 0.0) から反時計回りにまわり，(1.0, 0.6)，(0.5, 0.8)，(0.0, 0.8)，$\cdots$ と進み，最後のデータは始点と同じ (1.3, 0.0) で，13 点存在する．媒介変数の t は 0〜12 の整数とし，$x(t)$ と $y(t)$ の補間により，平面曲線を求めている．
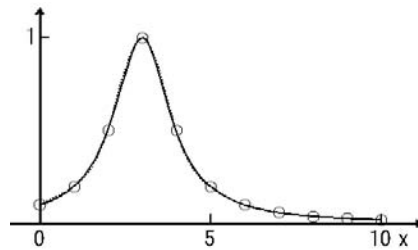
図 3.10　3 次の非周期スプラインによる補間．関数は $f(x) = \dfrac{1}{1+(x-3)^2}$ である．データ点と節点は同一で，1 刻みの 11 点 (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) であり，∘ 印で示されている．点線は補間曲線であり，実線は真の曲線である．(Z03_10_Spline3_IntNP.java)
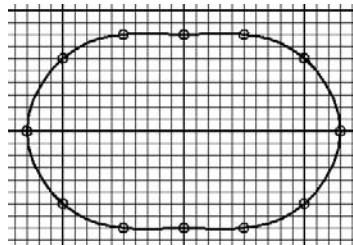
図 3.11　3 次の周期スプラインによる平面曲線の補間．データ点 (x, y) に，媒介変数 t を対応させ，$x(t)$ と $y(t)$ の補間により，平面曲線を求めている．∘ 印がデータ点および節点であり，実線が補間曲線である．(Z03_11_Spline3_IntP.java)

演習問題

3.1　下の表のデータを使って補間値のグラフを描けるように，プログラム Z03_03_lagrange.java を修正しなさい．

x	0.5	1.0	1.5	2.0	2.5	3.0	3.5
$\log x$	-0.693	0.000	0.405	0.693	0.916	1.098	1.252
x	4.0	4.5	5.0	5.5	6.0	6.5	
$\log x$	1.386	1.504	1.609	1.704	1.791	1.871	

解答例：ソースプログラム Q03_01_Z03_03_lagrange.java

3.2　B スプラインを使った補間のプログラム Z03_07_Bspline_Interpolation.java のデータ点と節点を変えて，補間の様子を調べなさい．

3.3　B スプラインを使った内挿のプログラム Z03_08_Bspline_Extended.java のデータ点と節点を変えて，補間の様子を調べなさい．

3.4　3 次の非周期スプライン補間のプログラム Z03_10_Spline3_IntNP.java のデータ点を変えて，補間の様子を調べなさい．

第 4 章

方程式の解法

代数方程式 $f(x) = 0$ の根を求めるとき，根の公式が存在する場合にはそれを使うことになるが，5 次以上の高次方程式では公式を作ることは不可能であるということが，19 世紀にアーベルとガロアによって独立に発見されている．解析的な解 (公式) が知られていない方程式の場合は，解となる x の値を反復法や試行錯誤で求めることになる．ある区間で単調で連続な関数 $f(x)$ の方程式 $f(x) = 0$ に解が 1 個あることがわかっているときには，2 分法，挟み撃ち法，ニュートン・ラフソン法，逆 2 次関数法などが役に立つ．複数の解や特異点が存在する可能性がある場合には，これらの方法をうまく組み合わせて解を探していくことになる．

4.1 代数方程式

2 次方程式には根の公式があり，3 次方程式にはカルダノ公式，4 次方程式にはフェラリ公式があるが，5 次以上の高次方程式の解の公式は存在しない．ここでは，2 次方程式の根の公式と 3 次方程式のカルダノ公式を扱う．

4.1.1 2 次方程式

2 次方程式の根の公式は，高等学校の数学で真っ先に覚えさせられる公式であり，いまさら解説の必要はないが，プログラミング言語の実例として取り上げることにする．2 次方程式を

$$ax^2 + bx + c = 0 \tag{4.1}$$

とすると，根の公式は

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{-b \pm \sqrt{d}}{2a} \qquad \text{ただし } d = b^2 - 4ac \tag{4.2}$$

である．係数 a, b, c の値によっていろいろの場合があるので，この式をそのままプログラムの中に書けばよいのは数式処理ができる特殊な言語だけであり，一般のプログラミング言語では，きちんと場合分けをして書かなければならない．それは，

$$\begin{array}{llll} a \neq 0 & d > 0 & & 2\text{実根}：x = \dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a} \\ & d = 0 & & \text{等根}：x = \dfrac{-b}{2a} \\ & d < 0 & & 2\text{複素共役根}：x = \dfrac{-b \pm i\sqrt{|b^2 - 4ac|}}{2a} \\ a = 0 & b \neq 0 & & x = -\dfrac{c}{b} \\ & b = 0 & c \neq 0 & \text{不能（解なし）} \\ & & c = 0 & \text{不定（すべてが解）} \end{array} \tag{4.3}$$

である．2 実根を求める際に，b^2 に比べ $4ac$ の値が非常に小さい場合には，$b > 0$ なら $x_1 = b - \sqrt{b^2 - 4ac}$，$b < 0$ なら $x_1 = b + \sqrt{b^2 - 4ac}$ の計算で桁落ちの誤差が大きくなる．そこで，分子の有理化をして $x_1 = \dfrac{4ac}{b + \sqrt{b^2 - 4ac}}$ または $x_1 = \dfrac{4ac}{b - \sqrt{b^2 - 4ac}}$ とするか，あるいは，もう片方の解 x_2 を最初に求めてから，それを使って解 $x_1 = \dfrac{c}{ax_2}$ を求めるようにしなければならない．

NijiHouteishiki.java のクラスとその説明は以下のとおりである．

NijiHouteishiki.java のクラス：

```
 1 public class  NijiHouteishiki{
 2   public NijiHouteishiki_Answer
 3       nijiHouteishiki(double a, double b, double c) {
 4     NijiHouteishiki_Answer ans = new NijiHouteishiki_Answer();
 5     double x, d;
 6     if (a != 0) {
 7       b /= a;  c /= a;
 8       if (c != 0) {
 9         b /= 2; d = b * b - c;
10         if (d > 0) {
11           if (b > 0) {
12             x = -b - Math.sqrt(d);
13           } else {
14             x = -b + Math.sqrt(d);
15           }
16           ans.kon= "二実根";
17           ans.kai[0]= x;
18           ans.kai[1]= c/x;
19           System.out.println("x = " + x + ", " + c / x);
20         } else if (d < 0) {
21           ans.kon= "二複素根";
22           ans.kai[0]= -b;
23           ans.kai[1]= Math.sqrt(-d);
24           System.out.println("x = " + (-b) + " +- "
25               + Math.sqrt(-d) + " i");
26         } else {
27           ans.kon= "重根";
28           ans.kai[0]= -b;
29           ans.kai[1]= 0;
```

```
30              System.out.println("x = " + (-b) + "(重解)");
31          }
32        } else {
33          ans.kon= "二実根";
34          ans.kai[0]= -b;
35          ans.kai[1]= 0;
36          System.out.println("x = " + (-b) + ", 0");
37        }
38      } else if (b != 0) {
39        ans.kon= "単一根";
40        ans.kai[0]= -c/b;
41        ans.kai[1]= 0;
42        System.out.println("x = " + (-c / b));
43      } else if (c != 0) {
44        ans.kon= "解なし";
45        ans.kai[0]= 0;
46        ans.kai[1]= 0;
47        System.out.println("解なし");
48      } else {
49        ans.kon= "解不定";
50        ans.kai[0]= 0;
51        ans.kai[1]= 0;
52        System.out.println("解不定");
53      }
54    return ans;
55    }
56 }
```

1. L1〜56: クラス NijiHouteishiki の定義.
2. L2, 3〜55: 戻り値が NijiHouteishiki_Answer 型のメソッド nijiHouteishiki の定義．a は 2 次方程式 $ax^2 + bx + c = 0$ の x^2 の項の係数，b は x の項の係数，c は定数項である.
3. L4: 2 次方程式の根を返すときに使うクラス NijiHouteishiki_Answer のインスタンス ans の作成.
4. L6: a がゼロかどうかの判定.
5. L7: ゼロでなかったら，2 次方程式であり，式を a で割って x^2 の項の係数が 1 である方程式 $x^2 + bx + c = 0$ の形式にする.
6. L8: c がゼロかどうかの判定.
7. L9: ゼロでなかったら，2 次方程式の判別式を d に作る．b を 2 で割り方程式を $x^2 + 2bx + c = 0$ の形式にする.
8. L10, 20, 26: 判別式が正か負かゼロかで，それぞれの処理をする.
9. L11: 判別式が正であったら，桁落ちを防ぐ処理のために b の正負を調べる.
10. L12: b が正であったら，解 $-b - \sqrt{d}$ の計算をする.
11. L14: b が正でなかったら，解 $-b + \sqrt{d}$ の計算をする.
12. L16: ほかで使うために NijiHouteishiki_Answer クラスのインスタンス ans の文字列 kon に“二重根”を代入する.
13. L17: ans の配列 kai[0] に根の x を代入する.
14. L18: ans の配列 kai[1] にもう一つの根 c/x を代入する.

15. L19: 根の情報をモニターに出力する.
16. L21〜25: 判別式が負であったら,“二複素根”の計算と出力をする.
17. L27〜30: 判別式がゼロであったら,“二重根”の計算と出力をする.
18. L33〜36: c がゼロであったら,“二実根”(ゼロと 1 実根) の計算と出力をする.
19. L39〜42: a がゼロで b がゼロでなければ,“単一根”の計算と出力をする.
20. L44〜47: a と b がゼロで c がゼロでなければ,“解なし”の出力をする.
21. L49〜52: a と b と c がゼロであるならば,“解不定”の出力をする.
22. L54: NijiHouteishiki_Answer クラスのインスタンス ans を返す.

NijiHouteishiki_Answer.java のクラスとその説明は以下のとおりである.

NijiHouteishiki_Answer.java のクラス:

```
1 public class NijiHouteishiki_Answer {
2   public String kon;
3   public double[] kai= new double[2];
4 }
```

1. L1〜4: クラス NijiHouteishiki_Answer の定義の始まり. nijiHouteishiki からの戻り値に使う. このクラスはメソッドの定義を含んでおらず, Fortran の構造体に似ている.
2. L2: 根の種類を返すための文字列の宣言をする.
3. L3: 2 根の数値を返すための配列を作る.

4.1.2 3 次方程式 (カルダノ公式)

3 次方程式の一般型は

$$x^3 + bx^2 + cx + d = 0 \tag{4.4}$$

とすることができる. 左辺の変曲点を 0 に移すように, 変換 $x = x' - \dfrac{b}{3}$ を使うと式 (4.4) は 2 次の項がない式

$$x'^3 + 3px' + q = 0 \quad \text{ただし } p = \frac{c}{3} - \frac{b^2}{9},\ \ q = d - \frac{bc}{3} + \frac{2b^3}{27} \tag{4.5}$$

にすることができる. さらに, $x' = u + v$ を代入して

$$u^3 + v^3 + 3uv(u + v) + 3p(u + v) + q = 0 \tag{4.6}$$

とする. 次に, $u^3 + v^3 = -q$ と置くと, $uv = -p$ となり, u, v は連立方程式

$$\left.\begin{aligned} u^3 + v^3 &= -q \\ u^3 v^3 &= -p^3 \end{aligned}\right\} \tag{4.7}$$

から求めることができる. これは, 2 次方程式の根と係数の関係になっているから, u^3 と v^3 は 2 次方程式

$$t^2 + qt - p^3 = 0 \tag{4.8}$$

の 2 根である. それらは,

$$
\left.
\begin{aligned}
t_1 &= \frac{1}{2}\left(-q + \sqrt{q^2 + 4p^3}\right) = u^3 \\
t_2 &= \frac{1}{2}\left(-q - \sqrt{q^2 + 4p^3}\right) = v^3
\end{aligned}
\right\} \tag{4.9}
$$

である．そこで，3 次方程式の根 x は

$$
\left.
\begin{aligned}
x_1 &= \sqrt[3]{t_1} + \sqrt[3]{t_2} - \frac{b}{3} \\
x_2 &= -\frac{1}{2}\left(\sqrt[3]{t_1} + \sqrt[3]{t_2}\right) + \frac{i\sqrt{3}}{2}\left(\sqrt[3]{t_1} - \sqrt[3]{t_2}\right) - \frac{b}{3} \\
x_3 &= -\frac{1}{2}\left(\sqrt[3]{t_1} + \sqrt[3]{t_2}\right) - \frac{i\sqrt{3}}{2}\left(\sqrt[3]{t_1} - \sqrt[3]{t_2}\right) - \frac{b}{3}
\end{aligned}
\right\} \tag{4.10}
$$

として求めることができる．根の性質は式 (4.9) の判別式 $q^2 + 4p^3$ で分類することになる．

❒ $q^2 + 4p^3 > 0$
t_1, t_2 は実数であり，$\sqrt[3]{t_1}$，$\sqrt[3]{t_2}$ として t_1, t_2 の実の立方根をとると，x_1 は実根であるが x_2, x_3 は共役な複素根となる．

❒ $q^2 + 4p^3 < 0$
t_1, t_2 は共役な複素根であり，それらの立方根の和は実数，差は虚数となるので，x_1 だけではなく x_2, x_3 も実根となる．立方根はそれぞれ 3 種類あるが，実際のプログラムでは極形式を使って，偏角が $\frac{1}{3}$ の立方根から求めるように統一してある．

❒ $q^2 + 4p^3 = 0$
方程式 (4.8) の解は等根 $t_1 = t_2$ であり，x_2, x_3 も実の等根となる．

4.2 節で使う方程式 $(x+1.3)^3-4.0 = 0$ に当てはめると，3 根 0.5214，$-2.2107\pm0.6987i$ が求められる．

プログラム (SanjiHouteishiki.java) のメソッド (SanjiHouteishiki) の一部と，その説明は以下のとおりである．

SanjiHouteishiki.java のメソッド (SanjiHouteishiki) の一部：

```
1   public Complex[] sanjiHouteishiki(double b ,double c, double d) {
2     Complex[] ans= new Complex[3];
3     Complex zero = new Complex(0.0, 0.0);
4   後略
```

1. L1〜: 戻り値が Complex 型配列のメソッド sanjiHouteishiki の定義である．引数の b は 3 次方程式の x^2 の項の係数，c は x の項の係数，d は定数項である．
2. L2: 戻り値のための Complex 型配列 ans のインスタンスを作成する．
3. L3: Complex 型のインスタンス zero を作成し，複素数のゼロを定義する．
4. L4: 以下は，本文での解説のとおりである．

4.2　2 分法

関数 $y = f(x)$ は 2 点 $x_{\min}$ と $x_{\max}$ で異符号であり，その 2 点間では連続で，方程式 $f(x) = 0$ に解があることがわかっているものとする．解を求めるには，まず，両端の座標値 $x_{\min}$，$x_{\max}$ を変数 x_{low}，x_{up} に代入して，それぞれの関数値 $f(x_{\text{low}})$ $f(x_{\text{up}})$ を求める．その 2 点の間に解があるのだから，$f(x_{\text{low}})f(x_{\text{up}}) < 0$ である．次に両端の中点の値を変数 x_{mid} に代入し，その点を解の推定値として関数値 $f(x_{\text{mid}})$ を求める．$f(x_{\text{mid}})$ が $f(x_{\text{low}})$ と同符号，$f(x_{\text{low}})f(x_{\text{mid}}) > 0$ ならば，解は x_{mid} と x_{up} の間にあるから， x_{mid} の値を変数 x_{low} に代入し，異符号，$f(x_{\text{low}})f(x_{\text{mid}}) < 0$ ならば，解は x_{low} と x_{mid} の間にあるのだから， x_{mid} の値を変数 x_{up} に代入する．同じ操作を繰り返して x_{mid} の値が収束条件の精度以内で定まったら，それを解とする．図 4.1 は 2 分法で関数 $f(x) = (x + 1.3)^3 - 4.0 = 0$ の解を -1.0 と 1.0 の間で求めるときの収束過程を示すものである．実線の曲線は関数であり，最上の直線は最初の $x_{\min}$ と $x_{\max}$ の位置を結んだものであり，中央の $*$ 印は x_{mid} の位置である．それより下の直線は，順に x_{low} と x_{up} の位置を結んだもので，10 回まで示してあり，収束の仕方がわかる．2 分法を 10 回進めたことで解の精度は約 3 桁，$(x_{\max} - x_{\min})/1000$ になっている．

2 分法により解を 1 個だけ探すことのできるプログラム (Nibunhou.java) のメソッド (nibunhou) と，その説明は以下のとおりである．

Nibunhou.java のメソッド (nibunhou)：

```
 1  public double nibunhou(double xmin, double xmax, MyFunction f,
 2      double eps) {
 3    double xlow, xup, xmid;
 4    double ylow, yup, ymid;
 5    eps= eps*(xmax-xmin);
 6    xlow= xmin;
 7    xup= xmax;
 8    xmid= 0.0;
 9    ylow= f.function(xlow);
10    yup= f.function(xup);
```
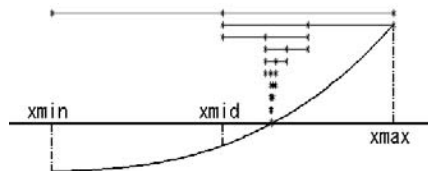
図 4.1　2 分法で解を求めるときの収束過程．実線の曲線は関数であり，最上の直線は最初の $x_{\min}$ と $x_{\max}$ の位置を結んだもの，以下順に x_{low} と x_{up} の位置を結んだもので，10 回まで示してある．

```
11      if (ylow*yup > 0) {
12        System.out.println("ERROR: ymin*ymax > 0!");
13        return 0.0;
14      }
15      for (int i=0; i<30; i++) {
16        xmid= (xup+xlow)/2;
17        ymid= f.function(xmid);
18        if (ylow*ymid > 0) {
19          xlow= xmid;
20          ylow= ymid;
21        } else {
22          xup= xmid;
23          yup= ymid;
24        }
25        if (Math.abs(xup-xlow)<eps) break;
26      }
27      return xmid;
28    }
```

1. L1, 2〜28: メソッド nibunhou の定義である．引数の xmin, xmax は解を探す範囲の下限と上限，f はインタフェース MyFunction クラスのインスタンスであり，解を求めるメソッドが定義されている．eps は解の精度の閾値で (xmax−xmin) に対する相対値である．戻り値は解である．
2. L5: (xmax−xmin) に対する相対値で与えられた閾値 eps の値を，相対値でなく絶対値にする．
3. L6〜10: xlow, xup, xmid, ylow, yup に初期値を代入する．
4. L11〜14: ylow*yup>0 であったら解がないかもしれないから，警告を出して終了する．
5. L15〜26: 2 分法を最大 30 回繰り返す．
6. L16, 17: xmid, ymid に新しい 2 分点での値を代入する．
7. L18: ylow*ymid の正負を調べる．
8. L19, 20: 正であったら，解は xup 側にあるから，xmid, ymid の値を xlow, ylow に代入する．
9. L22, 23: 正でなかったら，解は xlow 側にあるから，xmid, ymid の値を xup, yup に代入する．
10. L25: 解が存在する範囲 xup−xlow が eps 以下になったらループを抜け出す．
11. L27: 解として xmid を戻す．

4.3　挟み撃ち法

この方法はレギュラ・ファルシ法ともいわれる．2 分法では解の推定値として両端 x_0, x_1 の中点

$$x_2 = \frac{1}{2}(x_0 + x_1) \tag{4.11}$$

を使ったが，挟み撃ち法では，両端 x_0, x_1 での関数値の座標点を直線で結び，その直線と x 軸との交点

$$
\begin{aligned}
x_2 &= \frac{x_1 f(x_0) - x_0 f(x_1)}{f(x_0) - f(x_1)} \\
&= \frac{1}{2}(x_0 + x_1) + \frac{x_0 - x_1}{f(x_0) - f(x_1)} \frac{f(x_0) + f(x_1)}{2}
\end{aligned}
\tag{4.12}
$$

を解の推定値 x_2 とする方法である．式 (4.12) の第 2 右辺の第 1 項は 2 分法の項であり，第 2 項が近似を高める補正項となっている．この場合は，x_0 か x_1 のどちらかで，関数値が $f(x_2)$ と異符号である．それを x_1 であるとする．この操作

$$
x_{i+1} = \frac{x_i f(x_1) - x_1 f(x_i)}{f(x_1) - f(x_i)} \tag{4.13}
$$

を反復して根に近づくことになる．$\Delta x_1 = x_1 - a$, $\Delta x_i = x_i - a$ と置いて，解 a での $f(x)$ のテーラー展開

$$
\begin{aligned}
y_0 &= f(a)\Delta x_1 + \frac{f''(a)}{2!}\Delta x_1^2 + \cdots \\
y_i &= f(a)\Delta x_i + \frac{f''(a)}{2!}\Delta x_i^2 + \cdots
\end{aligned}
\tag{4.14}
$$

を使うと，式 (4.13) から

$$
x_{i+1} \simeq a + \Delta x_1 \Delta x_i \frac{f''(a)}{2f'(a)} \tag{4.15}
$$

となる．さらに a を移行すると

$$
x_{i+1} - a = \Delta x_{i+1} \simeq \Delta x_1 \Delta x_i \frac{f''(a)}{2f'(a)} \tag{4.16}
$$

となるから，Δx_1 が $\left| \Delta x_1 \frac{f''(a)}{2f'(a)} \right| < 1$ となる程度小さくなっていれば $\Delta x_{i+1} < \Delta x_i$ であり，この反復は収束することになる．図 4.2 は挟み撃ち法で関数 $f(x) = (x + 1.3)^3 - 4.0$ の解を -1.0 と 1.0 の間で求めるときの収束過程を示すものである．実線の曲線は関数であり，点線は次の予測値 x_2 とその次の予測値 x_3 を求める 1 次式である．最上の直線は最初の x_0 と x_1 の位置を結んだもの，以下順に x_2 と x_1，x_3 と x_1 の位置を結
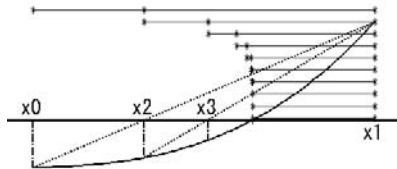
図 4.2　挟み撃ち法で解を求めるときの収束過程．太い実曲線は関数であり，点線は次の予測値 x_2 とその次の予測値を求める 1 次式である．最上の細い実直線は最初の x_0 と x_1 の位置を結んだもの，以下順に x_2 と x_1，x_3 と x_1 の位置を結んだもので，10 回まで示してある．

んだものであり，10 回まで示してあるので収束していく過程がわかる．10 回進めたところで解の精度は約 4 桁になっており，2 分法より 1 桁良い．

この例の場合は，挟み撃ちの右側の点は常に x_1 であったが，$[x_0, x_1]$ の範囲内に変曲点がある関数の場合には，両側の点が移動することもある．また，解となる点 a で $f'(a) = 0$ である関数の場合は，先の収束条件を満たさないので，解は求められない．

挟み撃ち法により解を 1 個だけ探すことのできるプログラム (Hasamiuchi.java) のメソッド (hasamiuchi) と，その説明は以下のとおりである．

Hasamiuchi.java のメソッド (hasamiuchi)：

```
 1  public static double hasamiuchi(double xmin, double xmax,
 2    MyFunction f, double eps) {
 3    eps= eps*(xmax-xmin);
 4    double xnxt=0.0, fmin, fmax, fnxt;
 5    fmin = f.function(xmin);
 6    fmax = f.function(xmax);
 7    if (fmin*fmax > 0) {
 8      System.out.println("ERROR: fmin*fmax > 0!");
 9      return 0.0;
10    }
11    for (int i=0; i<30; i++) {
12      xnxt = (xmin*fmax-xmax*fmin)/(fmax-fmin);
13      fnxt = f.function(xnxt);
14      if (Math.abs(xnxt-xmin)<eps || Math.abs(xmax-xnxt)<eps) break;
15      if (fnxt == 0) {
16        break;
17      } else if (fmin*fnxt > 0) {
18        xmin = xnxt;
19        fmin = fnxt;
20      } else {
21        xmax = xnxt;
22        fmax = fnxt;
23      }
24    }
25    return xnxt;
26  }
```

1. L1, 2〜26: メソッド hasamiuchi の定義である．引数の xmin，xmax は解を探す範囲の下限と上限，f はインタフェース MyFunction クラスのインスタンスであり，解を求めるメソッドが定義されている．eps は解の精度の閾値で (xmax−xmin) に対する相対値である．戻り値は解である．
2. L3: (xmax−xmin) に対する相対値で与えられた閾値 eps の値を，相対値から絶対値にする．
3. L5, 6: fmin，fmax に初期値を代入する．
4. L7〜10: fmin∗fmax>0 であったら解がないかもしれないから，警告を出して終了する．
5. L11〜24: 挟み撃ち法を最大 30 回繰り返す．

6. L12, 13: xnxt, fnxt に新しい値を代入する．
7. L14: xnxt が xmin あるいは xmax に eps より近づいたら，それが解であるから，ループを抜ける．
8. L15, 16: fnxt がゼロであったら，xnxt が解であるから，ループを抜ける．
9. L17〜19: fmin*fnxt が正であったら，解は xmax 側にあるから，xnxt, fnxt の値を xmin, fmin に代入する．
10. L21, 22: 正でなかったら，解は xmin 側にあるから，xnxt, fnxt の値を xmax, fmax に代入する．
11. L25: 解として xnxt を戻す．

4.4　反復法

解を求める方程式 $f(x)=0$ を変形して $x=g(x)$ としたとき，x のある値 x_0 から出発して，演算

$$x_{i+1}=g(x_i) \quad (i=0,\,1,\,\cdots) \tag{4.17}$$

を繰り返すと，$\lim\limits_{i\to\infty} x_i$ が a に収束すれば，その収束値が方程式 $f(x)=0$ の解 a である．a を式 (4.17) に代入すると $a=g(a)$ であるから，$y=g(x)$ と置いて $x=a$ でテーラー展開すると，

$$y-a=g'(a)(x-a)+\frac{g''(a)}{2!}(x-a)^2+\cdots \tag{4.18}$$

である．途中の推定値 x が a に近づいており $|x-a|$ が小さければ，$|g'(a)|<1$ のときには収束することになる．そこで，$|g'(a)|<1$ となるような関数 $g(x)$ を，元の関数 $f(x)$ から作ることができればよいことになる．式 (4.18) で，$g'(a)\neq 0$ の場合は x に関する 1 次の反復である．また，m 階以下の微分はゼロであり，m 階の微分になって初めてゼロでなくなる場合には，m 次の反復となる．先に述べた挟み撃ち法は 1 次の反復にあたる．

4.4.1　ニュートン・ラフソン法

関数 $f(x)$ とその導関数 $f'(x)$ が与えられているときに，方程式 $f(x)=0$ の解を求める方法である．まず，適当な出発点 x_0 を与え，座標 $(x_0,\,f(x_0))$ で，関数 $f(x)$ に接する接線と x 軸との交点

$$x_1=x_0-\frac{f(x_0)}{f'(x_0)} \tag{4.19}$$

を解の推定値として求める．さらに，収束するまで

$$\lim_{i\to\infty} x_i=x_{i-1}-\frac{f(x_{i-1})}{f'(x_{i-1})} \tag{4.20}$$

を繰り返すと解が求められる．導関数 $f'(x)$ が解析的にわかっている場合には，それを使えばよいが，わかっていない場合には関数 $f(x)$ を数値的に微分することでもよい．すなわち，Δx を適当に小さくとって，次の近似

$$f'(x_0) = \frac{f(x_0+\Delta x) - f(x_0-\Delta x)}{2\Delta x} \tag{4.21}$$

を使うこともできる．

この場合，式 (4.20) は

$$y = g(x) = x - \frac{f(x)}{f'(x)} \tag{4.22}$$

である．この式をテーラー展開の式 (4.18) に当てはめることにする．関数 $g(x)$ の 1 次導関数と 2 次導関数は，

$$\begin{aligned} g'(x) &= 1 - \frac{f'(x)f'(x) - f(x)f''(x)}{(f'(x))^2} = \frac{f(x)f''(x)}{(f'(x))^2} \\ g''(x) &= \frac{f(x)f'''(x)}{(f'(x))^2} + \frac{f'(x)f''(x)}{(f'(x))^2} - 2\frac{f(x)(f''(x))^2}{(f'(x))^3} \end{aligned} \tag{4.23}$$

であるから，$g'(a) = 0$，$g''(a) = \dfrac{f''(x)}{f'(x)}$ を使うと，$f'(a) \neq 0$ (a が重根でない) ならば，

$$y - a = \frac{1}{2}\frac{f''(a)}{f'(a)}(x-a)^2 + \cdots \tag{4.24}$$

となる．これは x に関する 2 次の反復であるから収束は速い．しかし，$f'(a) = 0$，$f''(a) \neq 0$ である場合には

$$\begin{aligned} \lim_{x\to a} g'(x) &= \frac{\lim_{x\to a} f(x)f''(x)}{\lim_{x\to a}(f'(x))^2} = \frac{\lim_{x\to a}\{f'(x)f''(x) + f(x)f'''(x)\}}{\lim_{x\to a} 2f'(x)f''(x)} \\ &= \frac{1}{2} + \frac{1}{2}\frac{\lim_{x\to a} f(x)f'''(x)}{\lim_{x\to a} f'(x)f''(x)} \\ &= \frac{1}{2} + \frac{1}{2}\frac{\lim_{x\to a}\{f'(x)f'''(x) + f(x)f''''(x)\}}{\lim_{x\to a}\{(f''(x))^2 + f'(x)f'''(x)\}} = \frac{1}{2} \end{aligned} \tag{4.25}$$

となるので $(x-a)$ の項が残ることになり，展開は

$$y - a = \frac{1}{2}(x-a) + \cdots \tag{4.26}$$

となって，1 次の反復に戻ってしまう．

図 4.3 は関数 $f(x) = (x+1.3)^3 - 4.0$ の解を，1.0 を始点としてニュートン・ラフソン法で求めるときの収束過程を示すものである．実線の曲線は関数であり，破線は次の予測値 x_1 とその次の予測値 x_2 を求める 1 次式である．最上の細い実直線は最初の x_0 と x_1 の位置を結んだもの，以下順に x_1 と x_2，x_2 と x_3 の位置を結んだもので，6 桁収束する 5 回まで示してある．
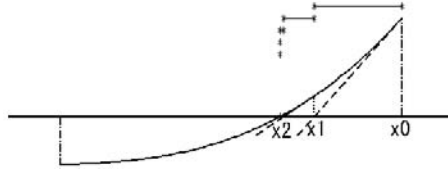
図 4.3　ニュートン・ラフソン法で解を求めるときの収束過程．実線の曲線は関数であり，破線は次の予測値 x_1 とその次の予測値 x_2 を求める 1 次式である．最上の細い実直線は最初の x_0 と x_1 の位置を結んだもの，以下順に x_1 と x_2，x_2 と x_3 の位置を結んだものである．6 桁収束する 5 回まで示してある．

挟み撃ち法でもニュートン・ラフソン法でも，予測値はともに近似 1 次式がゼロとなる値であるが，ニュートン・ラフソン法の場合には予測値の式 (4.22) $g(x) = x - \dfrac{f(x)}{f'(x)}$ の 1 階微分がゼロになるようになっているから，収束が速いのである．しかし，初期値の x_0 の選び方が良くないと発散してしまうので，注意が必要である．

ニュートン・ラフソン法により単調な関数の方程式の解を 1 個探すことのできるプログラム (NewtonRaphson.java) のメソッド (newtonRaphson) と，その説明は以下のとおりである．

NewtonRaphson.java のメソッド (newtonRaphson)：

```
 1  public static double newtonRaphson(double x,
 2    MyFunction f, MyFunction df, double eps) {
 3    double fn, dfn;
 4    for (int i=0; i<30; i++) {
 5      fn = f.function(x);
 6      if (Math.abs(fn) <= eps) break;
 7      dfn = df.function(x);
 8      x = x-fn/dfn;
 9    }
10    return x;
11  }
```

1. L1, 2〜11: メソッド newtonRaphson の定義である．引数の x は解を探す初期値，f はインタフェース MyFunction クラスのインスタンスであり，解を求める関数が定義されている．df では同じく導関数が定義されている．eps は解の関数値の精度の閾値である．戻り値は解である．
2. L4〜9: ニュートン・ラフソン法を最大 30 回繰り返す．
3. L5: x における関数値 fn を計算する．
4. L6: fn の絶対値が eps より小さくなったら，ループを抜ける．
5. L7: 導関数の値 dfn を計算する．
6. L8: ニュートン・ラフソン法の次の予測値を計算し，ループを繰り返す．
7. L10: 解の x を戻す．

4.4.2　反復法の一般論

反復法の一般論を使うと，ニュートン・ラフソン法の場合の予測値の式 (4.22) は，解 a においてその 1 階微分がゼロになるように作られている，ということがわかる．反復法は，解を求める方程式 $f(x) = 0$ を変形して $x = g(x)$ とし，x のある値 x_0 から出発して，式 (4.17) を繰り返し，ある値 a に収束すれば，その収束値が方程式 $f(x) = 0$ の解であるとする方法である．$x = g(x)$ の解は $y = x$ と $y = g(x)$ の交点であるから，反復の過程を図に描くと，図 4.4 左となる．この図の例では x_0 から始めて $x_1, x_2, x_3, \cdots$ という具合に交点 a に向かって収束していく．一般的に関数 $g(x)$ が交点を中心として x 軸方向から $\pm 45°$ の範囲に収まっている場合に収束することがわかる．収束をより速くするためには，図 4.4 右のように，$g(x)$ が交点で x 軸に平行，すなわち，$g'(a) = 0$ となるように選ぶことである．

ニュートン・ラフソン法では，そのようにするため

$$g(x) = x + h(x)f(x) \tag{4.27}$$

として $h(x)$ を決める．それには，$g'(a) = 0$ とするための条件

$$\lim_{x \to a} g'(x) = 1 + h(a)f'(a) = 0 \tag{4.28}$$

から，$h(x) = -\dfrac{1}{f'(x)}$ とすればよいことがわかる．これを式 (4.27) に代入したものが式 (4.22) である．
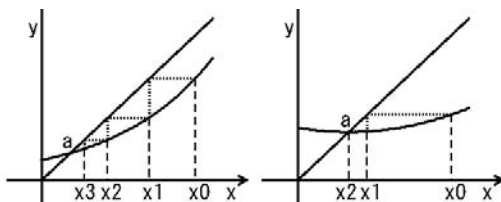
図 4.4　方程式 $f(x) = 0$ を変形した $x = g(x)$ で解を求める反復の過程．x_0 から始めて $x_1, x_2, \cdots$ と交点 a に向かって収束していく．右図はニュートン・ラフソン法に対応しており，収束が速い．

4.5　逆 2 次関数法

一般的に，曲線を近似するには，1 次式より 2 次式のほうが良い．挟み撃ち法は，2 点間を結ぶ 1 次式で推定値を決めているが，ここでは 2 次式を使うことにする．すると，3 点が必要であるから，2 分法の中間点も使うことになる．この場合は普通の内挿とは異なり，$y = 0$ となる x の推定値 $x_{\rm int}$ を内挿で求めるのであるから，逆関数

の 2 次式 $x = x(y)$ を使うことになる．そこで，解をある程度追いつめてからでないと，$x_{\rm int}$ が両端の値 $x_{\rm min}$ と $x_{\rm max}$ の外側になってしまうことがあるので，内側に収まるようになる程度までは 2 分法などを使う必要がある．図 4.5 は 2 分法などで使った関数の解を，逆 2 次関数法で求めていく過程である．最初の逆 2 次関数のグラフは，$x_{\rm max}$ より右側に飛び出している部分もあるが，$x_{\rm int}$ は内側に納まっているので，この結果を使うことができる．2 回目ですでに解を求める関数を良く近似していることがわかる．

逆 2 次関数法により，解を 1 個だけ探すことのできるプログラム (Gyaku_2jikansuu.java) のメソッド (gyaku_2jikansuu) と，その説明は以下のとおりである．

Gyaku_2jikansuu.java のメソッド (gyaku_2jikansuu)：

```
 1  public double gyaku_2jikansuu(double xmin, double xmax,
 2      MyFunction f, double eps) {
 3    double xint, xmid, yint, ymid;
 4    eps= eps*(xmax-xmin);
 5    double ymin= f.function(xmin);
 6    double ymax= f.function(xmax);
 7    if (ymin*ymax > 0) {
 8      System.out.println("ERROR: ymin*ymax > 0!");
 9      return 0.0;
10    }
11    xmid= (xmax+xmin)/2;
12    ymid= f.function(xmid);
13    for (int i=0; i<30; i++) {
14      xint= xmin*ymid*ymax/(ymin-ymid)/(ymin-ymax)
15        + xmid*ymax*ymin/(ymid-ymax)/(ymid-ymin)
16        + xmax*ymin*ymid/(ymax-ymin)/(ymax-ymid);
17      yint= f.function(xint);
18      if (xint > xmid) {
19        xmin= xmid;
20        ymin= ymid;
21      } else {
```
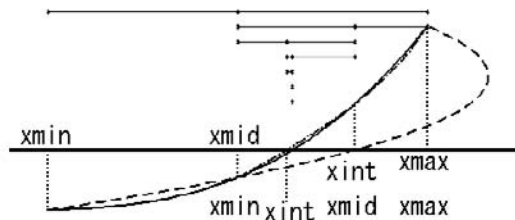
図 4.5　逆 2 次関数法で解を求めていく過程．実線は解を求める関数，破線は最初の逆 2 次関数，一点鎖線は 2 度目の逆 2 次関数である．収束は非常に速く，6 回で 14 桁精度がある解が求められる．

```
22          xmax= xmid;
23          ymax= ymid;
24        }
25        xmid= xint;
26        ymid= yint;
27        if (Math.abs(xmax-xmin)<eps) break;
28      }
29      return xmid;
30    }
```

1. L1, 2〜30: メソッド gyaku_2jikansuu の定義である．引数の xmin，xmax は解を探す範囲の下限と上限，f はインタフェース MyFunction クラスのインスタンスであり，解を求める関数が定義されている．eps は解の精度の閾値で (xmax−xmin) に対する相対値である．戻り値は解である．
2. L4: (xmax−xmin) に対する相対値で与えられた閾値 eps の値を，相対値から絶対値にする．
3. L5, 6: ymin，ymax に初期値を代入する．
4. L7〜10: ymin*ymax>0 であったら，解がないかもしれないから，警告を出して終了する．
5. L11, 12: xmid，ymid に最初の予測値を代入する．
6. L13〜28: 逆 2 次関数法を最大 30 回繰り返す．
7. L14〜17: 逆 2 次関数法で次の予測値を xint，yint に代入する．
8. L18〜20: xint>xmid であったら，解は xmax 側にあるから，xmid，ymid の値を xmin，ymin に代入する．
9. L22, 23: xint>xmid でなかったら，解は xmin 側にあるから，xmid，ymid の値を xmax，ymax に代入する．
10. L25, 26: xmid，ymid に xint，yint を代入する．
11. L27: 解が存在する範囲 xmax−xmin が eps 以下になったらループを抜け出す．
12. L29: 解として xmid を戻す．

4.6　多数の解がある場合の汎用プログラム

解を探す関数の性質がわかっている場合には，これまでに述べた方法をうまく使って解を求めることができるであろう．しかし，関数によっては，いろいろな条件によって解の個数が変化したり，発散する特異点が存在する場合もあるであろう．そのような場合には，片方の端から探し始め，特異点を避けながら解を探索するプログラムがあると便利である．関数を $f(x)$ とし，解を探索する範囲を $[x_{\rm low}, x_{\rm up}]$ とする．まず，$f(x_{\rm low})$ から始め，刻みを Δx として順次，点 $x_i\ (= x_{\rm low} + i \times \Delta x)$ での $f(x_i)$ の値を求めていく．もし，$f(x_{i-1})f(x_i) < 0$ となったら，x_{i-1} と x_i の間で 2 分法や逆 2 次関数法などを使って，解を追いつめていく．もし，$|f(x_{\rm mid})|$ が $|f(x_{i-1})|$ と $|f(x_i)|$ より大きかったら，x_{i-1} と x_i は発散する特異点を挟んでいる可能性がある．そこで，2 分法と同様に $f(x_{\rm mid})$ を次々と求め，適当な回数 (6〜10 回) にわたり $|f(x_{\rm mid})|$ の増

大が続いたら，関数の逆数 $\dfrac{1}{f(x)}$ のゼロ点を見つけて，それを特異点とする．その後，$f(x_{i+1})$ を求めて $f(x_i)$ と比較して先に進む．もし，$f(x_{i-1})f(x_i) > 0$ であっても，微係数が $\dfrac{f(x_i) - f(x_{i-1})}{f(x_i)} > 0$，かつ $\dfrac{f(x_{i-1}) - f(x_{i-2})}{f(x_{i-1})} < 0$ であったら，x_{i-2} と x_i の間に解が 2 個あるかもしれないから，その間をより細かな刻み，例えば $\Delta x/5$ を指定して再帰呼び出しを行う．この方法では，3 重根は単根との区別が付けられないので，さらなる改良が望まれる．

図 4.6 は関数

$$f(x) = \frac{(x+1)x(x-2)(x-3)(x-4)}{x-1.1} - 2.4 \tag{4.29}$$

と，プログラム FindZero で求めた解をプロットした図である．特異点 1.1 と互いに近接した 2 根 2.3816 と 2.4527 もとらえられている．

プログラム (FindZero.java) のメソッド (findZero) の引数と戻り値は以下のとおりである．

FindZero.java のメソッド (findZero) の引数と戻り値：

```
1   public double[] findZero(double xmin, double xmax, double dx,
2       MyFunction f, double eps) {
```

1. メソッド findZero の定義である．
2. xmin は解を探し始める x の最小値 (始点) である．
3. xmax は解を探す x の最大値 (終点) である．
4. dx は解を探すときの x の刻みである．
5. f はインタフェース MyFunction のインスタンスであり，解を求める関数が定義されている．
6. eps は解の精度の閾値である．
7. 戻り値は解の配列であり，要素の数は求められた解の数である．
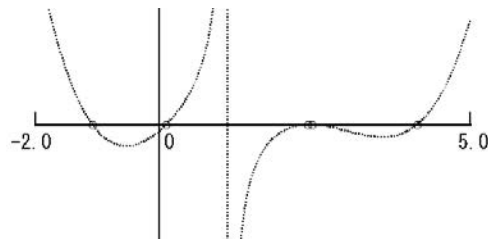
図 4.6　関数のグラフ (点線) とプログラム FindZero で求めた複数の解 (○)．特異点や極めて近接する解があるように，関数は $\dfrac{(x+1)x(x-2)(x-3)(x-4)}{x-1.1} - 2.4$ とする．5 個の解があり，それらは -1.076，0.1014，2.3816，2.4527，4.1404 である．縦の点線は特異点 1.1 である．(Z04_06_FindZero.java)

プログラム (FindZero.java) のメソッド (findZero) を使うときの注意事項は以下のとおりである.

```
 1 import mypackage.FindZero;
 2 import mypackage.MyFunction;
 3 中略
 4 FindZero fz= new FindZero();
 5 Function f= new Function();
 6 中略
 7 class Function implements MyFunction{
 8     public double function(double x) {
 9     return ここに式 f(x) を書く;
10   }
11 }
```

1. L1, 2: クラス FindZero と MyFunction をインポートする. ここでは, 直下のディレクトリ [mypackage] にあることとしている.
2. L4, 5: メソッド findZero を使うメインメソッドなどの中で, 両クラスのインスタンス [fz] と [f] を作る. その後メソッド findZero や function を使う.
3. L7〜11: インタフェース MyFunction を実装するクラス [Function] を定義する.
4. L8〜10: メソッド function をオーバーライドする.
5. L9: 実際の関数の式を定義する.
6. [ ] 内の名前は, 固定されておらず, 任意である.

演習問題

4.1　ニュートン・ラフソン法による解法のプログラム NewtonRaphsonTest.java を修正して, 関数 $-x^3+3x^2-2x-1=0$ の解を求めなさい.

答： -0.3247

4.2　複数解の解法汎用プログラムの Z04_06_FindZero.java を使って, $\sin x^2=0$ の解を $[0,4]$ の範囲で求めなさい.

答： 0, 1.772, 2.507, 3.070, 3.545, 3.963

第 5 章

数値積分法

ある関数の積分が解析的にわかっている場合には，その関数の定積分は容易に求められる．しかし，積分が解析的にわかっていない場合には，数値計算で近似的に求めることになる．その数値積分の基本となる方法は，定積分がわかっている関数 $P(x)$ を使って被積分関数 $f(x)$ を近似し，その定積分の値を近似値とする方法である．その近似関数は多くの場合多項式である．

5.1 矩形近似

関数 $f(x)$ をある区間 $[x_0, x_1]$ で積分したいとき，関数値がある 1 点 x_a でしかわからない場合には，積分値を $(x_1 - x_0)f(x_a)$ で近似して我慢せざるを得ない．もし，ある区間 $[x_0, x_n]$ を n 等分した各区間の中点で $f(x)$ の値を知ることができれば，積分値を

$$h\sum_{i=0}^{n-1} f\left(\frac{x_i + x_{i+1}}{2}\right), \quad h = x_1 - x_0 \tag{5.1}$$

で近似することができる．等間隔でない場合には，それぞれの幅を重み $h_i = x_{i+1} - x_i$ として

$$\sum_{i=0}^{n-1} h_i f\left(\frac{x_i + x_{i+1}}{2}\right), \quad h_i = x_{i+1} - x_i \tag{5.2}$$

となる．これを矩形近似という．採用する関数値を，中点ではなく各間隔の左端の点あるいは右端の点での値とすることも考えられる．

図 5.1 は積分値

$$\int_0^1 (x+1)^3 \mathrm{d}x \tag{5.3}$$

を 4 分割の矩形近似で求める例である．左図からそれぞれ左端，中央，右端の関数値を使う場合であり，一番右の図は次節で扱う台形近似の場合である．積分の真値は 1.0 であるが，左図からそれぞれ 0.730，0.932，1.168，0.949 である．精度は関数型によってまちまちであろうが，一般的に中央の点を使う場合や台形近似が良いであろう．
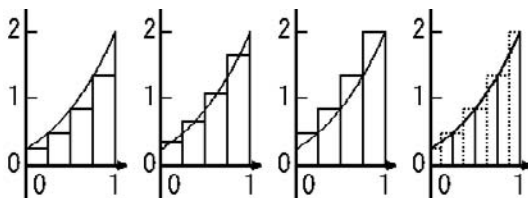
図 5.1　矩形近似．定積分は $\int_0^1 (x+1)^3 \mathrm{d}x$ であり，左図から左端，中央，右端の関数値を使った場合である．一番右の図の実線は台形近似での図であり，点線は公式を変形したときの，矩形である．

5.2　ニュートン・コーツ型積分

区間 $[\,x_0, x_n\,]$ を n 等分した $n+1$ 個の点において関数値 $f(x_i)$ が使える場合の近似法である．刻み幅を $h=(x_n-x_0)/n$ と置くと，分点は $x_i = ih + x_0$ $(0 \leq i \leq n)$ であり，すべての分点 x_i における $f(x_i)$ の値を使うことにする．

5.2.1　台形近似

最も単純な 2 点 $(n=1)$ の場合には，2 点間を 1 次式

$$
\begin{aligned}
L_1(x) &= \frac{f(x_1)-f(x_0)}{x_1-x_0}(x-x_0)+f(x_0) \\
&= f(x_0)\frac{x-x_1}{x_0-x_1}+f(x_1)\frac{x-x_0}{x_1-x_0}
\end{aligned}
\tag{5.4}
$$

で近似する．この場合，積分される領域の形は台形となっている[1]ので台形近似といわれる．定積分は式 (5.4) を積分して求めることができるが，台形の面積の公式を使うと

$$
\int_{x_0}^{x_1} f(x)\mathrm{d}x = \{f(x_0)+f(x_1)\}h/2 = h\left\{\frac{1}{2}f(x_0)+\frac{1}{2}f(x_1)\right\} \tag{5.5}
$$

である．この台形では h が高さとなっている．最後の式は，見方を変えると幅 h の領域の両端の高さ $f(x_0)$ と $f(x_1)$ に，重み 1/2 ずつを掛け合わせて加え，幅 h を掛けたものであるともいえる．この重みは，合計が 1 であれば，どのような組み合わせでもよいが，1/2 ずつの重みが順当であろう．分点が $m+1$ 点あり間隔はすべて等しく h である場合に，台形公式を繰り返し適用すると，積分公式は

$$
\int_{x_0}^{x_m} f(x)\mathrm{d}x = h\left\{\frac{1}{2}f(x_0)+\sum_{i=1}^{m-1} f(x_i)+\frac{1}{2}f(x_m)\right\} \tag{5.6}
$$

となる．この式は台形公式といわれる．図 5.1 の一番右の図は台形公式での積分の様子である．実線は元の台形公式そのままの台形の寄せ集めであるが，点線は台形公式

[1] 台形といっても，台は横に寝かされている．

を繰り返し適用したときの公式であり，両端のデータ点における重み幅が $\frac{1}{2}$ で，途中の重み幅が 1 の矩形の寄せ集めで，式 (5.6) となっていることを示している．

5.2.2 シンプソンの公式

次は等間隔の 3 点（x_0, x_1, x_2）を使う 2 次式の場合である．2 次式は

$$L_2(x) = a(x-x_1)^2 + b(x-x_1) + c \tag{5.7}$$

とすることができる．$x_2 - x_1 = x_1 - x_0 = h$ を使うと，

$$\begin{aligned} f(x_0) &= ah^2 - bh + c \\ f(x_1) &= c \\ f(x_2) &= ah^2 + bh + c \end{aligned} \tag{5.8}$$

である．これから $a = \dfrac{f(x_0) - 2f(x_1) + f(x_2)}{2h^2}$，$b = \dfrac{f(x_2) - f(x_0)}{2h}$，$c = f(x_1)$ が決まる．a, b, c を式 (5.7) に代入して整理すると，補間法の式 (3.5)

$$L_2(x) = \sum_{i=0}^{2} \left(f(x_i) \prod_{j=0\,(j\neq i)}^{2} \frac{x - x_j}{x_i - x_j} \right) \tag{5.9}$$

である．他方，a, b, c を式 (5.7) に代入し，x_1 が 0 になるまで関数を平行移動すると，定積分をより容易に求めることができて，それは

$$\int_{x_0}^{x_2} f(x)\mathrm{d}x = h\left\{ \frac{1}{3}f(x_0) + \frac{4}{3}f(x_1) + \frac{1}{3}f(x_2) \right\} \tag{5.10}$$

となる．台形近似の式 (5.5) と比べて，この場合は中点 x_1 での重みがより大きくなっている．この近似は 2 次式を使って導いているから，被積分関数が 2 次式までは正確であり，誤差が生じない．

分点が $2m+1$ 点あり，間隔が h である場合にこの公式を繰り返し適用すると，積分公式は

$$\int_{x_0}^{x_{2m}} f(x)\mathrm{d}x = \frac{h}{3}\left\{ f(x_0) + f(x_{2m}) + 4\sum_{j=1}^{m} f(x_{2j-1}) + 2\sum_{j=1}^{m-1} f(x_{2j}) \right\} \tag{5.11}$$

となる．この公式はシンプソンの公式といわれる．

プログラム (integral_Simpson.java) のメソッド (integral_Simpson) と，その説明は以下のとおりである．

integral_Simpson.java のメソッド (integral_Simpson)：

```
1   public double integral_Simpson(int n, double xmin, double xmax,
2         MyFunction f) {
3     if (n==0) return 0.0;
4     double x;
5     double h= (xmax-xmin)/n;
```

```
 6      double sum= f.function(xmin);
 7      for (int i=1; i<n; i+=2) {
 8        x= xmin+i*h;
 9        sum += 4.0*f.function(x)+2*f.function(x+h);
10      }
11      sum -= f.function(xmax);
12      return sum*h/3.0;
13    }
```

1. L1, 2〜13: メソッド integral_Simpson の定義である．引数 n は x 座標の分割数であり，偶数でなければならない．xmin は積分領域の最小値，xmax は積分領域の最大値である．f はインタフェース MyFunction クラスのインスタンスであり， 解を求める関数が定義されている．戻り値は積分値である．
2. L5: 変数 h に x 座標の分割の間隔を代入する．
3. L6: 積分値を足し込んでいく変数に，初期値として積分領域の最小値での関数値を代入する．
4. L7〜10: 2 刻みを 1 単位として積分を処理するので，ループの数は n/2 となる．
5. L8: 変数 x に次の 2 刻みの小さいほうの値を代入する．
6. L9: x の値での関数値は 4 倍し，次の刻み x+h での関数値は 2 倍した値を sum に足し込む．
7. L11: ループが終わると，最後の x の値，xmax における関数値は 2 倍が足し込まれているから，ここで，1 倍分を引き去っておく．
8. L12: sum に間隔 h を掛け，3 で割って積分値として返す．

図 5.2 の左右の図の実線は積分 $\int_0^{2\pi} -\frac{x\sin x}{2\pi}\mathrm{d}x$ の被積分関数である．左図の破線は台形公式，右図の点線はシンプソンの公式を使っており，ともに $n=4$ のときの近似関数である．

次に，n の数を増やすと，どのように近似が良くなっていくかを調べる．図 5.3 は積分 $\int_0^{2\pi} -\frac{x\sin x}{2\pi}\mathrm{d}x = 1$ の台形公式 ($\times$) とシンプソンの公式（$\circ$）による近似値が，分割数 n の増加とともに，どのように収束していくかを調べたものである．

図 5.2 は $n=4$ の場合であるから図 5.3 の $n=4$ に対応している．表 5.1 には近似計算値を示してある．$n=2$ では両方とも積分値がゼロになっているが，これは関数値が両端と中点でゼロになっているからである．図 5.2 からも，シンプソンの公式のほうが優れており，台形公式では収束が遅いことがわかる．数値的に理解するために，表 5.1 にいくつかの分割数 n についての近似値を載せてある．$n=20$ での精度は，台形公式での計算では 3 桁しかないが，シンプソンの公式によると 5 桁はある．そこで，第 6 章で述べるルンゲ・クッタ法などの微分方程式を，等間隔の分点で解いた場合などに，解の関数を規格化する積分などでは，シンプソンの公式がよく使われている．
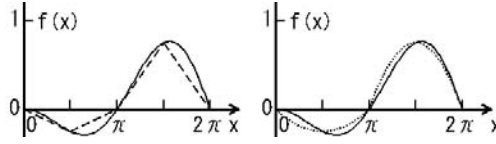
図 5.2　関数 $\left\{-\dfrac{x\sin x}{2\pi}\right\}$ の台形近似とシンプソンの近似．左図の破線は台形公式，右図の点線はシンプソンの公式を使っており，ともに $n=4$ の近似関数である．
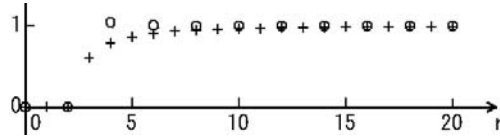
図 5.3　台形公式とシンプソンの公式による近似値の収束性．+ 印は台形公式，○ 印はシンプソンの公式で求めた値である．積分は $\displaystyle\int_0^{2\pi} -\frac{x\sin x}{2\pi}\mathrm{d}x$ であり，真値は 1 である．横軸は分割数 n である．シンプソンの公式では 1 区間で 2 分点を使うので，分割数 n が偶数のときのみ計算されている．

表 5.1　台形公式とシンプソンの公式で求めた積分値．積分は $\displaystyle\int_0^{2\pi} -\frac{x\sin x}{2\pi}\mathrm{d}x$ であり，真値は 1 である．n は分割数である．

n	台形公式	シンプソンの公式
2	0.0	0.0
4	0.7853	1.0471
6	0.9068	1.0076
20	0.9917	1.00005

5.2.3　一般型

n 次の補間多項式は，ラグランジュの補間公式 (3.6) によると

$$L_n(x)=\sum_{i=0}^{n}\left(f(x_i)\prod_{j=0\,(j\neq i)}^{n}\frac{x-x_j}{x_i-x_j}\right) \tag{5.12}$$

である．ここで，関数 $\Pi(x)$ を

$$\Pi(x)=\prod_{i=0}^{n}(x-x_i) \tag{5.13}$$

で定義する．これを使うと式 (5.12) は

$$L_n(x)=\Pi(x)\sum_{i=0}^{n}\frac{f(x_i)}{(x-x_i)\Pi'(x_i)} \tag{5.14}$$

となる．ここでは，

$$\begin{aligned}\Pi'(x) &= \sum_{i=0}^{n}\prod_{j\neq i}^{n}(x-x_j) \\ \Pi'(x_i) &= \prod_{j\neq i}^{n}(x_i-x_j)\end{aligned} \tag{5.15}$$

を使っている．次に $x = x_0 + th\,(0 \geq t \geq n)$ と置くと，$x - x_i = h(t-i)$ であるから，

$$\Pi(x) = h^{n+1}t(t-1)\cdots(t-n) \tag{5.16}$$

であり，

$$\begin{aligned}\Pi'(x_i) &= h^n i(i-1)\cdots 2\cdot 1\cdot(-1)\cdot(-2)\cdots(i-n) \\ &= h^n(-1)^{n-i}i!(n-i)!\end{aligned} \tag{5.17}$$

となる．これを使うと式 (5.14) は

$$L_n(x) = \sum_{i=0}^{n} f(x_i)\frac{(-1)^{n-i}}{i!(n-i)!}\,\frac{t(t-1)\cdots(t-n)}{t-i} \tag{5.18}$$

となる．そこで，定積分は

$$\begin{aligned}\int_{x_0}^{x_n} L_n(x)\mathrm{d}x &= \sum_{i=0}^{n} f(x_i)\frac{(-1)^{n-i}}{i!(n-i)!}h\int_0^n \frac{t(t-1)\cdots(t-n)}{t-i}\mathrm{d}t \\ &= h\sum_{i=0}^{n} w_i f(x_i)\end{aligned} \tag{5.19}$$

となる．最後の式で重み

$$w_i = \frac{(-1)^{n-i}}{i!(n-i)!}\int_0^n \frac{t(t-1)\cdots(t-n)}{t-i}\mathrm{d}t \tag{5.20}$$

を使っている．$n = 1, 2, 3$ の場合の w_i を表 5.2 に示す．

ある関数を一定の範囲 $[\,x_0,\, x_n\,]$ で積分するなら，分点の数を多くするほうが精度が良くなるのは当然であるが，全体の分点数 n と間隔 h を一定にしたままで積分するのならば，多数の分点を使う高次の多項式を使って計算するより，式 (5.11) のように 2 次式のシンプソンの近似を繰り返し使うほうが，簡単であるし精度も良いようである．

表 5.2　ニュートン・コーツ型積分公式で使われる重み．

n	$w_i\ (i = 0, \cdots, n)$	
1	1/2，1/2	台形公式
2	1/3，4/3，1/3	シンプソンの公式
3	3/8，9/8，9/8，3/8	

5.3　ガウス型積分

先のニュートン・コーツ型積分では，前もって決めてある $n+1$ 個の分点を使って n 次式で近似したが，n 個の分点の位置座標 x_i と分点での重み w_i の n 個をうまく使えば，自由度は $2n$ になるので，$2n-1$ 次式で近似することができる．そこで，前もって分点でのデータが与えられているのではなく，任意の点において関数値を計算することができる場合には，この方法が有利である．区間 $[a,b]$ において，重み関数[2] $w(x)$ を付けた関数 $f(x)$ の定積分を検討する．ある n について，積分公式

$$\int_a^b w(x)f(x)\mathrm{d}x = \sum_{i=1}^{n} w_i f(x_i) \tag{5.21}$$

が，$(2n-1)$ 次の多項式まで正確に成り立つように，データ点 (ここでは分点というよりふさわしいであろう) x_i と重み w_i を決定すればよい．ニュートン・コーツ型の公式では両端の点を使っているから，和は $i=0$ からであったが，式 (5.21) 以下では $i=1$ からである．$f(x)$ を $(2n-1)$ 次の多項式であるとする．最高次の項の係数が c_n である n 次の多項式

$$S_n(x) = c_n(x-x_1)(x-x_2)\cdots(x-x_n) = c_n\prod_{i=1}^{n}(x-x_i) \tag{5.22}$$

で，$f(x)$ を割った商と剰余をそれぞれ $Q_{n-1}(x)$，$R_{n-1}(x)$ とすると，

$$f(x) = S_n(x)Q_{n-1}(x) + R_{n-1}(x) \tag{5.23}$$

となり，$Q_{n-1}(x)$ と $R_{n-1}(x)$ は高々 $(n-1)$ 次の多項式である．一方，$S_n(x_i)=0$ であるから，$R_{n-1}(x_i)=f(x_i)$ となることを考慮すると，式 (5.14) と類似の式

$$R_{n-1}(x) = S_n(x)\sum_{i=1}^{n}\frac{f(x_i)}{(x-x_i)S_n'(x_i)} \tag{5.24}$$

となることがわかる．これらを式 (5.21) に使うと，

$$\int_a^b w(x)S_n(x)Q_{n-1}(x)\mathrm{d}x + \sum_{i=1}^{n}\left\{\frac{f(x_i)}{S_n'(x_i)}\int_a^b \frac{w(x)S_n(x)}{x-x_i}\mathrm{d}x\right\} = \sum_{i=1}^{n} w_i f(x_i) \tag{5.25}$$

となる．ここで，任意の $f(x)$ に対して，すなわち，任意の $Q_{n-1}(x)$ に対して

$$\int_a^b w(x)S_n(x)Q_{n-1}(x)\mathrm{d}x = 0 \tag{5.26}$$

---

[2]. 後で出てくるチェビシェフの積分公式では，重み関数は $w(x)=\dfrac{1}{\sqrt{1-x^2}}$ である．被積分関数が $\dfrac{1}{\sqrt{1-x^2}}$ の形に変数変換をすることができる部分を含んでいる場合に，その関数を $w(x)f(x)$ として，積分公式を $f(x)$ に当てはめることになる．一般的に，積分範囲が有限であり，被積分関数の発散点を含まなければ，重み関数は必要でなく，$w(x)\equiv 1$ である．

となるようにすれば，式 (5.25) から重み w_i は

$$w_i = \frac{1}{S'_n(x_i)} \int_a^b \frac{w(x)S_n(x)}{x - x_i} \mathrm{d}x \tag{5.27}$$

として決まる．$(n-1)$ 次の任意の多項式 $Q_{n-1}(x) = \sum_{i=0}^{n-1} a_i x^i$ に対して式 (5.26) が成り立つためには，

$$\int_a^b w(x)S_n(x)x^k \mathrm{d}r = 0 \quad (k = 0, 1, \cdots, n-1) \tag{5.28}$$

が成り立つように $S_n(x)$ を選べばよい．区間 $[a, b]$ において，重み関数を $w(x)$ として式 (5.28) が成り立つような直交多項式系 $\psi_0(x), \psi_1(x), \cdots, \psi_n(x), \cdots$ は特殊関数論ですでにわかっている．そこで，$S_n(x) = \psi_n(x)$ と置けること，および $S_n(x) = c_n \prod_{i=1}^{n} (x - x_i)$ であることから，データ点 $\{x_1, \cdots, x_n\}$ は $\psi_n(x) = 0$ の実数解であることになる．

5.3.1　区間が $[-1, 1]$ で重み関数が $w(x) = 1$ の場合

区間が $[-1, 1]$ で重み関数が $w(x) = 1$ の場合には，ガウス・ルジャンドルの公式になる．この場合使われる直交多項式系はルジャンドル多項式であり，その n 次の多項式は

$$P_n(x) = \frac{1}{2^n n!} \left(\frac{\mathrm{d}}{\mathrm{d}x}\right)^n (x^2 - 1)^n \tag{5.29}$$

である．データ点はこの n 次多項式の n 個の実数解であり，重み w_i は式 (5.27) から

$$w_i = \frac{1}{P'_n(x_i)} \int_{-1}^{1} \frac{P_n(x)}{x - x_i} \mathrm{d}x \tag{5.30}$$

であり，定積分は処理が可能で，

$$w_i = \frac{2(1 - x_i^2)}{(n+1)^2 \{P_{n+1}(x_i)\}^2} \tag{5.31}$$

となる．この積分公式はガウス・ルジャンドルの積分公式といわれる．

❐ $[-1, 1]$ で $n = 1$ の場合

$P_1(x) = x$ だから，$x_1 = 0$，$w_1 = 2$ である．$2n - 1 = 1$ だから，1 次式までは誤差が入らない．図 5.4 の左図は 1 次式の積分 $\int_{-1}^{1} \left(\frac{x}{2} + 1\right) \mathrm{d}x$ である．

❐ $[-1, 1]$ で $n = 2$ の場合

$P_2(x) = \frac{1}{2}(3x^2 - 1)$ だから，$(x_1 = -\frac{1}{\sqrt{3}}, w_1 = 1), (x_2 = \frac{1}{\sqrt{3}}, w_2 = 1)$ である．$2n - 1 = 3$ だから，3 次式までは誤差が入らない．図 5.4 の右図は 3 次式の積分 $\int_{-1}^{1} \frac{(x+1)^3}{4} \mathrm{d}x$ である．
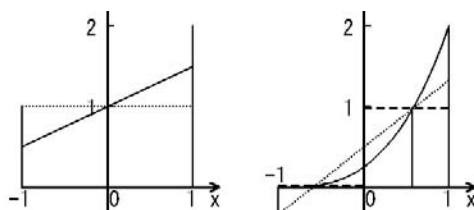
図 5.4 ガウス・ルジャンドルの積分 $n=1,2$ の例．左図は $n=1$ でデータ点は 0.0，重みは 2.0，被積分関数は 1 次式 $0.5x+1$ (実線) である．近似積分値は $w_0 f(x_0)=2.0\times 1.0=2.0$ で誤差はない．右図は $n=2$ でデータ点は $\left(-\frac{1}{\sqrt{3.0}}, \frac{1}{\sqrt{3.0}}\right)$，重みは (1.0, 1.0) であり，被積分関数は 3 次式 $\frac{(x+1)^3}{4}$ (実線) である．近似積分値は $\sum_{i=0}^{1} w_i f(x_i)=1.0\times 0.0189\cdots +1.0\times 0.9811\cdots=1.0$ でやはり誤差はない．

左図の 1 次式の場合には，中央をデータ点として，重みを 2 とすれば正しい積分値が得られるのは自明であるが，右図の場合には，データ点は 2 点しか使わず，重みは (1.0, 1.0) であるのに，任意の 3 次式の積分に誤差が入らないことは一見不思議な気がする．これまでの議論で示されているように，データ点の値を，3 次式の積分までできるように，2 次のルジャンドル多項式のゼロ点 $\pm\frac{1}{\sqrt{3.0}}$ としたからである．

表 5.3 は データ点が ($n=10$) のガウス・ルジャンドル積分公式のデータ値である．重みの合計は $\sum_{i=1}^{10} w_i=2.0$ である．データ点は中央が粗く，両端で密になっていることを見るために，図 5.5 には，データ点の位置に重みの大きさを破線で示している．また，重みは各データ点が受け持つ横の領域の幅であるともいえるから，その幅を実線で示している．

実際の積分 $\int_a^b f(x')\mathrm{d}x'$ では積分範囲が $[a,b]$ であるから，ガウス・ルジャンドルの積分公式 $\int_{-1}^{1} f(x)\mathrm{d}x$ に当てはめるために，変数変換

表 5.3 ($n=10$) のガウス・ルジャンドル積分公式のデータ値．n が偶数であるから，データ点の絶対値が同じ正負の座標値について重みは同じ値となる．

座標値	重 み
± 0.148874338981631211	0.295524224714752870
± 0.433395394129247191	0.269266719309996355
± 0.679409568299024406	0.219086362515982044
± 0.865063366688984511	0.149451349150580593
± 0.973906528517171720	0.066671344308688138
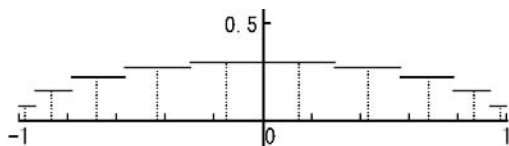
図 5.5　ガウス・ルジャンドル積分公式（$n=10$）のデータ値．データ点の位置に重みの大きさが破線で示されている．実線は，重みを各データ点が受け持つ横の領域の幅と解釈したときのその幅である．合計は 2 である．

$$x' = \frac{(b-a)}{2}x + \frac{(a+b)}{2}$$
$$\mathrm{d}x' = \frac{(b-a)}{2}\mathrm{d}x \qquad (5.32)$$

が必要である．これを使うと，実際の積分公式は

$$\frac{(b-a)}{2}\int_{-1}^{1} f\left(\frac{(b-a)}{2}x + \frac{(a+b)}{2}\right)\mathrm{d}x \qquad (5.33)$$

である．

プログラム (Gauss_Legendre.java) のコンストラクタとメソッド (gauss_Legendre) 等の一部と，その説明は以下のとおりである．

Gauss_Legendre.java のコンストラクタ (Gauss_Legendre) と
メソッド (gauss_Legendre) 等の一部：

```
 1 public class Gauss_Legendre {
 2   private double[] s,w;
 3   private int m;
 4   public Gauss_Legendre(int m) {
 5   中略
 6       final double s10[] = {  0.148874338981631211,
 7         0.433395394129247191, 0.679409568299024406,
 8         0.865063366688984511, 0.973906528517171720};
 9       final double w10[] = {  0.295524224714752870,
10         0.269266719309996355, 0.219086362515982044,
11         0.149451349150580593, 0.066671344308688137 6};
12   中略
13     this.m= m;
14     if (m==3) {
15       s= s3; w= w3;
16     } else if (m==10) {
17       s= s10; w= w10;
18   中略
19     }
20   }
21
22   public double gauss_Legendre(double xmin, double xmax, MyFunction f) {
23     double x;
```

```
24     int j;
25     double sum = 0.0;
26     double a= (xmax-xmin)/2.0;
27     double b= a+xmin;
28     if (m/2*2 == m) {
29       int mm= m/2;
30       for (int i=(mm-1); i>=0; i--) {
31         x= -a*s[i]+b;
32         sum += w[i] * f.function(x);
33       }
34       for (int i=0; i<mm; i++) {
35         x= a*s[i]+b;
36         sum += w[i] * f.function(x);
37       }
38     } else {
39   中略
40     }
41     return a*sum;
42   }
43 }
```

1. L1〜43: クラス Gauss_Legendre の定義である．
2. L2: コンストラクタとメソッドで共通に使うデータ用配列 s, w の宣言をする．
3. L3: コンストラクタとメソッドで共通に使うデータ点の数 m の宣言をする．
4. L4〜20: コンストラクタ Gauss_Legendre の定義である．引数 m は使うデータ点の数．
5. L6〜11: 使うデータ点の数が 10 点の場合のデータ点の x 座標値と重みである．
6. L13: 引数 m の値をこのクラスで共通に使うフィールド m に代入する．
7. L17: m=10 の場合に，このクラスで共通に使うデータ用配列 s, w に，それぞれ L6, L9 から始まるデータ用配列の参照を代入する．
8. L20: コンストラクタ Gauss_Legendre の定義の終了．
9. L22〜42: メソッド gauss_Legendre の定義である．引数 xmin は積分領域の最小値，xmax は積分領域の最大値である．f はインタフェース MyFunction クラスのインスタンスで，解を求める関数が定義されている．戻り値は積分値である．
10. L25: 積分値を足し込んでいく変数 sum をゼロで初期化する．
11. L26, 27: x 座標を標準座標系に座標変換する係数を決定する．
12. L28: データ点の数が偶数の場合と奇数の場合で処理が異なるので，それを判定する．
13. L29: 偶数の場合の処理が始まる．変数 mm にデータ点の数の半分を代入する．
14. L30〜33: x の負側の処理を，絶対値の大きい左側から始め，積分値を sum に加える．
15. L34〜37: x の正側の処理を，小さいほうから始め，積分値を sum に加える．
16. L41: sum に座標変換の係数 a を掛け，積分値として返す．
17. L42: メソッド gauss_Legendre の定義の終了．
18. L43: クラス Gauss_Legendre の定義の終了．

5.3.2　区間が $[-1,1]$ で重み関数が $w(x)=\dfrac{1}{\sqrt{1-x^2}}$ の場合

重み関数が $w(x)=\dfrac{1}{\sqrt{1-x^2}}$ の場合の直交多項式系はチェビシェフ多項式であり，その n 次の多項式は

$$T_n(x)=\cos(n\cos^{-1}x) \tag{5.34}$$

である．$n \leq 4$ の具体的な式は次のとおりである．

$$\left.\begin{array}{lcl} T_0(x) &=& 1 \\ T_1(x) &=& x \\ T_2(x) &=& 2x^2-1 \\ T_3(x) &=& 4x^3-3x \\ T_4(x) &=& 8x^4-8x^2+1 \end{array}\right\} \tag{5.35}$$

データ点は

$$x_i=\cos\frac{(2i-1)\pi}{2n} \quad (i=1,\cdots,n) \tag{5.36}$$

なので単純である．重み w_i は i によらず一定で

$$w_i=\frac{\pi}{n} \tag{5.37}$$

となるので，表にしておく必要はない．そこで積分公式は

$$\int_{-1}^{1}\frac{f(x)}{\sqrt{1-x^2}}\mathrm{d}x=\frac{\pi}{n}\sum_{i=1}^{n}f\left(\cos\frac{2i-1}{2n}\pi\right) \tag{5.38}$$

である．この公式は，被積分関数が積分領域の両端 a, b において $\dfrac{1}{\sqrt{(x-a)(b-x)}}$ の発散をしている場合に使うことができる．例えば，積分

$$\int_0^2\frac{\mathrm{d}x}{(x^2-2x+2)\sqrt{x(2-x)}}=\frac{\pi}{2\sqrt{2}} \tag{5.39}$$

がそうである．図 5.6 は被積分関数 $\dfrac{1}{(x^2-2x+2)\sqrt{x(2-x)}}$ (実線) と，重み関数として $\dfrac{1}{\sqrt{x(2-x)}}$ を分離してプログラムに与える関数 $\dfrac{1}{(x^2-2x+2)}$ (点線) である．被積分関数自体には両端 $x=a, b$ において $\dfrac{1}{\sqrt{x}}$ 型の発散があるけれども，その発散は弱く積分 (面積) は有限値である．また，計算では，その発散を重み関数として分離するので，プログラムに与える関数が素直な関数となり，収束の速いことは表 5.4 からも理解できる．

プログラム (Gauss_Tchebycheff.java) のメソッド (gauss_Tchebycheff) と，その説明は以下のとおりである．
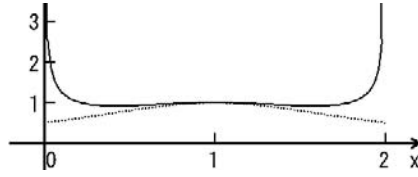
図 5.6 ガウス・チェビシェフ積分公式が使われる被積分関数．被積分関数そのものは $\dfrac{1}{(x^2-2x+2)\sqrt{x(2-x)}}$ で，実線で示されている．破線は，重み関数 $\dfrac{1}{\sqrt{x(2-x)}}$ を分離してプログラムに与える関数 $\dfrac{1}{(x^2-2x+2)}$ である．

表 5.4 ガウス・チェビシェフ積分の使用するデータ点の数と精度．積分は $\displaystyle\int_0^2 \frac{\mathrm{d}x}{(x^2-2x+2)\sqrt{x(2-x)}}$ であり，積分値は $\dfrac{\pi}{2\sqrt{2}}$ である．

データ点数	積分値	データ点数	積分値
真値	2.221441469079183	5	2.222102120831805
1	3.141592653589793	10	2.221441370870214
2	2.094395102393195	19	2.221441469079196
3	2.243994752564138	20	2.221441469079181
4	2.217594814298677	21	2.221441469079183

Gauss_Tchebycheff.java のメソッド (gauss_Tchebycheff)：

```
 1  public double gauss_Tchebycheff(int m, double xmin, double xmax,
 2          MyFunction f) {
 3    double a, b, x;
 4    double sum = 0.0;
 5    double pi= Math.PI;
 6    double w= pi/m;
 7    pi= pi/2/m;
 8    a= (xmax-xmin)/2.0;
 9    b= a+xmin;
10    for (int i=0; i<m; i++) {
11      x= a*Math.cos((2*i+1)*pi)+b;
12      sum += w * f.function(x);
13    }
14    return a*sum;
15  }
```

1. L1, 2〜15: メソッド gauss_Tchebycheff の定義である．引数の m はデータ点の数，xmin は積分領域の最小値，xmax は積分領域の最大値である．f はインタフェース MyFunction クラスのインスタンスであり，解を求める関数が定義されている．戻り値は積分値である．
2. L4: 積分値を足し込んでいく変数 sum を初期化する．

3. L8, 9: x 座標を標準座標系に座標変換する係数 a, b を決定する．
4. L10〜13: データ点の数だけ積分値を sum に足し込んでいく．
5. L14: sum に座標変換の係数 a を掛け，積分値として返す．

5.3.3　区間が $[0, \infty)$ の場合

重み関数を $w(x) = e^{-x}$ とすることができれば，ガウス・ラゲールの公式

$$\int_0^\infty e^{-x} f(x) \mathrm{d}x \cong \sum_{i=1}^{n} w_i f(x_i) \tag{5.40}$$

が使える．この場合の直交多項式系はラゲール多項式である．表 5.5 はデータ点の数 $n = 5$ のガウス・ラゲール積分公式の座標値 x_i，重み w_i と重み関数 $\exp(-x_i)$ の値である．データ点 x_i は 0.26，1.41，3.59，7.08，12.64 というように原点から離れるに従って間隔がどんどん大きくなっているが，重み w_i は小さくなり役割が減少するように見える．しかし，重み関数 $\exp(-x_i)$ の減少の仕方と比較すると減少が遅く，それなりの役割があることがわかる．

表 5.6 はデータ点の数 $n = 3$, 5, 10 の場合の，積分 $\displaystyle\int_0^\infty 4e^{-x}\frac{\sin x}{x}\mathrm{d}x$ の精度を示す表である．この積分の真値は π である．図 5.7 の実線は被積分関数 $e^{-x}\dfrac{\sin x}{x}$，破線は重み関数 e^{-x}，点線は計算に使う関数 $\dfrac{\sin x}{x}$ である．

重み関数を除いた関数 $\dfrac{\sin x}{x}$ は正負に振動しているけれども，重み関数 e^{-x} が掛

表 5.5　データ点の数（$n = 5$）のガウス・ラゲール積分公式のデータ点の座標値 x_i，重み w_i と重み関数 $\exp(-x_i)$ の値.

座標値 x_i	重み w_i	$\exp(-x_i)$
2.635603197181409e−1	5.21755610582809e−1	0.7683
1.41340305910651679	3.98666811083176e−1	0.2433
3.596425771040722	7.59424496817060e−2	0.0274
7.085810005858838	3.61175867992205e−3	8.3689E−4
1.264080084427578e1	2.33699723857762e−5	3.2372E−6

表 5.6　ガウス・ラゲールの積分の精度．積分は $\displaystyle\int_0^\infty 4e^{-x}\frac{\sin x}{x}\mathrm{d}x$ である．左の列はデータ点の数で，上から $n = 3$, 5, 10 であり，右の列は「計算値 − 真値 (π)」である．

n	計算値 − 真値
3	−0.014472164
5	0.000012664
10	0.000000116
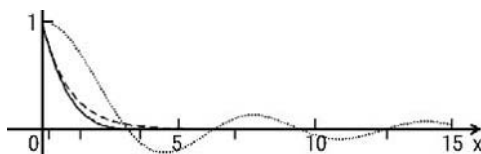
図 5.7 ガウス・ラゲール積分公式が使われる被積分関数，重み関数と計算に使う関数．実線は被積分関数 $e^{-x}\dfrac{\sin x}{x}$，破線は重み関数 e^{-x}，点線は計算に使う関数 $\dfrac{\sin x}{x}$ である．$n=5$ の場合のデータ点の位置が，x 軸から下に付けたバーで示されている．

かっているから早く減衰している．図 5.7 によると，$n=5$ でも被積分関数の関数値がゼロではないように見えるデータ点は 2 点しかないが，5 桁の有効数字がある．ちなみに，この積分の一般型は $\displaystyle\int_0^\infty e^{-ax}\frac{\sin bx}{x}\mathrm{d}x = \tan^{-1}\frac{b}{a}$ である．

実際の積分 $\displaystyle\int_a^\infty f(x')\mathrm{d}x'$ では積分範囲が $[a,\infty]$ であるから，ガウス・ラゲールの積分公式 $\displaystyle\int_0^\infty f(x)\mathrm{d}x$ を使う場合には，やはり変数変換

$$\begin{aligned} x' &= x-a \\ \mathrm{d}x' &= \mathrm{d}x \end{aligned} \tag{5.41}$$

が必要である．これを使うと，実際の積分公式は

$$\int_a^\infty f(x-a)\mathrm{d}x \tag{5.42}$$

である．

関数が x の無限大まで定義されており，e^{-x} で減衰している場合には，ガウス・ラゲールの公式が当てはまることになるが，x の有限領域で関数の変化が激しい場合には，その領域ではガウス・ルジャンドルの公式を繰り返し使い，変化が少なくなった減衰領域でガウス・ラゲールの公式を使うことになる．

プログラム (Gauss_Laguerre.java) のコンストラクタ (Gauss_Laguerre) とメソッド (gauss_Laguerre) の一部，およびその説明は以下のとおりである．

Gauss_Laguerre.java のコンストラクタ (Gauss_Laguerre) と
メソッド (gauss_Laguerre) の一部：

```
1 public class Gauss_Laguerre {
2   private double[] s,w;
3   private int m;
4   public Gauss_Laguerre(int m) {
5     final double s3[] = {4.1577455678348e-1,
6       2.294280360279, 6.2899450829375 };
7     final double w3[] = {7.11093009929e-1,
8       2.7851773356924e-1, 1.0389256501586e-2 };
9   中略
```

```
10     this.m= m;
11     if (m==3) {
12       s= s3; w= w3;
13   中略
14     }
15   }
16
17   public double gauss_Laguerre(double xmin, MyFunction f) {
18     double x;
19     double sum = 0.0;
20     for (int i=0; i<m; i++) {
21       x= s[i]+xmin;
22       sum += w[i] * f.function(x);
23     }
24     return sum;
25   }
26 }
```

1. L1〜26: クラス Gauss_Laguerre の定義である．
2. L2: コンストラクタとメソッドで共通に使うデータ用配列 s, w の宣言である．
3. L3: コンストラクタとメソッドで共通に使うデータ点の数 m の宣言である．
4. L4〜15: コンストラクタ Gauss_Laguerre の定義である．引数 m は使うデータ点の数である．
5. L5〜8: 使うデータ点の数が 3 点の場合のデータ点の x 座標値と重みである．
6. L10: 引数 m の値をこのクラスで共通に使うフィールド m に代入する．
7. L12: m=3 の場合に，このクラスで共通に使うデータ用配列 s, w に，それぞれ L5, L7 から始まるデータ用配列の参照を代入する．
8. L15: コンストラクタ Gauss_Laguerre の定義の終了．
9. L17〜25: メソッド gauss_Laguerre の定義である．xmin は積分領域の最小値，f はインタフェース MyFunction クラスのインスタンスであり，解を求める関数が定義されている．戻り値は積分値である．
10. L19: 積分値を足し込んでいく変数 sum をゼロ初期化する．
11. L20〜23: 積分値を sum に加える．
12. L24: sum を積分値として返す．
13. L25: メソッド gauss_Laguerre の定義の終了．
14. L26: クラス Gauss_Laguerre の定義の終了．

5.3.4 区間が $(-\infty, \infty)$ の場合

重み関数を $w(x) = e^{-x^2}$ とすることができれば，ガウス・エルミートの公式

$$\int_{-\infty}^{\infty} e^{-x^2} f(x)dx \cong \sum_{i=1}^{n} w_i f(x_i) \tag{5.43}$$

を使える．この場合の直交多項式系はエルミート多項式である．表 5.7 はデータ点の数 $n = 10$ のガウス・エルミート積分公式の座標値 x_i，重み w_i と重み関数 $\exp(-x_i^2)$ の値の表である．

表 5.7　データ点の数（$n = 10$）のガウス・エルミートの積分公式のデータ点の座標値 x_i，重み w_i，および重み関数 $\exp\left(-x_i^2\right)$ の値.

座標値 x_i	重み w_i	$\exp(-x_i^2)$
±0.342901327223705	6.108626337353258e−1	0.88907
±1.036610829789514	2.401386110823147e−1	0.34144
±1.756683649299882	3.387439445548106e−2	0.45687E−1
±2.532731674232790	1.343645746781233e−3	0.16372E−2
±3.436159118837738	7.640432855232621e−6	0.74508E−5

表 5.8　ガウス・エルミート積分の精度．被積分関数は $e^{-x^2}\cos x$ である．左の列はデータ点の数，上から $n = 10, 15$ であり，右の列は「計算値 − 真値 $\left(\sqrt{\pi}e^{-0.25}\right)$」である.

n	計算値 − 真値
10	−1.99840E−15
15	4.44089E−16
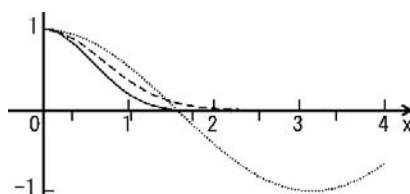
図 5.8　ガウス・エルミート積分公式が使われる被積分関数，重み関数と計算に使う関数．この場合は対称なので正の部分しか示していない．実線は被積分関数 $e^{-x^2}\cos x$，破線は重み関数 e^{-x^2}，点線は計算に使う関数 $\cos x$ である．$n = 10$ の場合のデータ点の位置が，x 軸から下に付けたバーで示されている.

表 5.8 はデータ点の数 $n = 10, 15$ の場合の，積分 $\int_{-\infty}^{\infty} e^{-x^2}\cos x\mathrm{d}x$ の精度である．この積分の真値は $\sqrt{\pi}e^{-0.25} = 1.38\cdots$ である.

実際の積分範囲は $(-\infty, \infty)$ であるが，この場合は対称なので図 5.8 には正の部分しか示していない．図の実線は被積分関数 $e^{-x^2}\cos x$，破線は重み関数 e^{-x^2}，点線は計算に使う関数 $\cos x$ である．ガウス・ラゲールの場合の重み関数は e^{-x} であったが，ガウス・エルミートの場合は e^{-x^2} であるので減衰はより速く，データ点 x_i もより小さい値に収まっている.

プログラム (Gauss_Hermit.java) のコンストラクタ (Gauss_Hermit) とメソッド (gauss_Hermit) の一部，およびその説明は以下のとおりである.

Gauss_Hermit.java のコンストラクタ (Gauss_Hermit) と
メソッド (gauss_Hermit) の一部：

```
 1 public class Gauss_Hermit {
 2   private double[] s,w;
 3   private int m;
 4   public Gauss_Hermit(int m) {
 5       final double s10[] = {3.429013272237046e-1,
 6       1.036610829789514, 1.756683649299882,
 7       2.532731674232790, 3.436159118837738 };
 8       final double w10[] = {6.108626337353258e-1,
 9       2.401386110823147e-1, 3.387439445548106e-2,
10       1.343645746781233e-3, 7.640432855232621e-6 };
11   中略
12     this.m= m;
13     if (m==10) {
14       s= s10; w= w10;
15   中略
16     }
17   }
18
19   public double gauss_Hermit(MyFunction Func) {
20     double x;
21     double sum = 0.0;
22     if (m/2*2 == m) {
23       int mm= m/2;
24       for (int i=mm-1; i>=0; i--) {
25         x= -s[i];
26         sum += w[i] * Func.function(x);
27       }
28       for (int i=0; i<mm; i++) {
29         x= s[i];
30         sum += w[i] * Func.function(x);
31       }
32   中略
33     }
34     return sum;
35   }
36 }
```

1. L1〜36: クラス Gauss_Hermit の定義である．
2. L2: コンストラクタとメソッドで共通に使うデータ用配列 s, w の宣言である．
3. L3: コンストラクタとメソッドで共通に使うデータ点の数 m の宣言である．
4. L4〜17: コンストラクタ Gauss_Hermit の定義．引数 m は使うデータ点の数である．
5. L5〜10: 使うデータ点の数が 10 点の場合のデータ点の x 座標値と重みである．
6. L12: 引数 m の値をこのクラスで共通に使うフィールド m に代入する．
7. L14: m=10 の場合に，このクラスで共通に使うデータ用配列 s, w に，それぞれ L5, L8 から始まるデータ用配列の参照を代入する．
8. L17: コンストラクタ Gauss_Hermit の定義の終了．

9. L19〜35: メソッド gauss_Hermit の定義である．引数の Func はインタフェース MyFunction クラスのインスタンスであり，解を求める関数が定義されている．戻り値は積分値である．
10. L21: 積分値を足し込んでいく変数 sum をゼロで初期化する．
11. L22: データ点の数が偶数の場合と奇数の場合で処理が異なるので，その判定をする．
12. L23: 偶数の場合の処理が始まる．変数 mm にデータ点の数の半分を代入する．
13. L24〜27: x の負側の処理を，絶対値の大きい左側から始め，積分値を sum に加える．
14. L28〜31: x の正側の処理を，小さいほうから始め，積分値を sum に加える．
15. L34: sum を積分値として返す．
16. L35: メソッド gauss_Hermit の定義の終了．
17. L36: クラス Gauss_Hermit の定義の終了．

5.4　誤差の評価

数値積分の誤差の評価を，重み関数 $w(x)$ がある一般的な場合で検討する．区間 $[a,b]$ において，重み関数 $w(x)$ を掛けた関数 $f(x)$ の定積分は，ある n 個の点における $f(x_i)$ の値の 1 次結合によって

$$\int_a^b w(x)f(x)\mathrm{d}x = \sum_{i=1}^{n} w_i f(x_i) + R \tag{5.44}$$

で近似することができる．この右辺第 1 項は積分の近似式で，w_i は関数 $f(x)$ に無関係な正の重み定数，第 2 項は誤差の項である．重み定数 w_i は次のようにして決めることができる．$f(x)$ が N 次までの多項式なら近似は正確な式となるので，誤差は $R=0$ となることを条件とする．このことは，$j \le N$ で

$$\int_a^b w(x)x^j\mathrm{d}x = \sum_{i=1}^{n} w_i x_i^j \tag{5.45}$$

が成り立つようにすることである．$N=n-1$ とすれば，この式で $j=0$ 〜 $(n-1)$ までの n 個の連立方程式ができるから，x_i がすべて決まっていれば n 個の w_i を決めることができる．これはニュートン・コーツ型積分公式である．また，$N=2n-1$ の場合には，$2n$ 個の連立方程式ができるから，すべての x_i も未知数として n 個の w_i とともに決めることになる．これはガウス型積分公式である．$f(x)$ が $(N+1)$ 階連続微分可能であるとして，点 c でテーラー展開すると

$$\begin{aligned} f(x) \quad &= \quad f(c) + \frac{f'(c)}{1!}(x-c) + \frac{f''(c)}{2!}(x-c)^2 + \\ &\quad \cdots + \frac{f^{(N)}(c)}{N!}(x-c)^N + \frac{f^{(N+1)}(\xi)}{(N+1)!}(x-c)^{(N+1)} \end{aligned} \tag{5.46}$$

である．これを式 (5.44) に代入して移行すると

$$R = \frac{1}{(N+1)!}\left\{\int_a^b w(x) f^{(N+1)}(\xi)(x-c)^{(N+1)}\mathrm{d}x - \sum_{i=1}^{n} w_i f^{(N+1)}(\xi_i)(x-c)^{(N+1)}\right\} \tag{5.47}$$

となる．積分領域での $\left|f^{(N+1)}(x)\right|$ の最大値を M と置き，テーラー展開をした点 c を積分領域の中心 $(a+b)/2$ にとると，全積分領域において $|x-c| \leq (b-a)/2$ であるから，上の式から

$$|R| \leq \frac{M(b-a)^{(N+1)}}{2^{N+1}(N+1)!}\left\{\int_a^b w(x)\mathrm{d}x + \sum_{i=1}^{n} w_i\right\} \tag{5.48}$$

である．式 (5.45) で $j=0$ を使うと

$$\int_a^b w(x)\mathrm{d}x = \sum_{i=1}^{n} w_i \tag{5.49}$$

であるから，最後に不等式

$$|R| \leq \frac{M(b-a)^{(N+1)}}{2^{N}(N+1)!}\int_a^b w(x)\mathrm{d}x \tag{5.50}$$

になった．これから誤差の限界は，n 点のデータを使うニュートン・コーツ型積分公式では n 階微分の項であるが，ガウス型積分公式では $2n$ 階微分の項 M になるので，積分領域を大きくとりすぎたりしなければ，非常に効率の良いことがわかる．

演習問題

5.1　$\displaystyle\int_0^\infty \frac{2}{x^2+1}\mathrm{d}x = \pi$ である．π の値を 3 桁求めなさい．

☞ ガウス・ルジャンドルの積分公式を複数回使う．

5.2　$\displaystyle\left(\int_0^\infty \frac{e^{-x}}{\sqrt{x}}\mathrm{d}x\right)^2$ の値を求めなさい．

☞ ガウス・ラゲールの積分公式を使う．

答：3.14

第 6 章

常微分方程式の解法

微分方程式ではなく普通の方程式の解を求めると，解のいくつかの値が決まり，その値をその方程式に代入すると，その方程式が成り立つことになる．他方，微分方程式を解くということは，解のいくつかの値を決めるのではなく，その関数をその微分方程式に代入すると，その微分方程式が成り立つような関数としての解を求めることである．

6.1 1階常微分方程式

1階常微分方程式

$$\frac{\mathrm{d}y(x)}{\mathrm{d}x} = f(x,\, y(x)) \tag{6.1}$$

の解 $y(x)$ を，初期条件

$$y(x_0) = y_0 \tag{6.2}$$

のもとで，数値計算によって求めることを検討する．この問題を積分形に変換すると，

$$y(x) = y_0 + \int_{x_0}^{x} f(t,\, y(t))\mathrm{d}t \tag{6.3}$$

となる．この式では積分変数を x ではなく t としている．いま，初期条件の $(x_0,\, y(x_0))$ から解を求め始めて，離散的な点の x_n までは $y_i = y(x_i)$ が求められているとする．次の点 $x_{n+1} = x_n + h$ での関数値は，式の上では

$$y_{n+1} = y_n + \int_{x_n}^{x_n+h} f(x,\, y(x))\mathrm{d}x \tag{6.4}$$

で求められる．しかし，x_n から先の x では $y(x)$ は決められていないのだから，$f(x,\, y(x))$ もまだわからない．そこで，このままでは，この積分を計算することはできない．この式に平均値の定理を使うと，

$$y_{n+1} = y_n + hf(x_n + \theta h,\, y(x_n + \theta h)) \quad (0 \leq \theta \leq 1) \tag{6.5}$$

となる．しかし，これでも $x > x_n$ での $y(x)$ の値 $y(x_n + \theta h)$ はまだわかっていないので使えない．

もし，関数 $f(x, y)$ が x のみの関数で，y に依存していなければ，$f(x)$ はわかっているので，式 (6.4) は普通の定積分である．$h = x_{n+1} - x_n$ を小さな刻みとして，台形公式を使えば，それは，

$$y_{n+1} = y(x_n + h) = y_n + h\left(\frac{1}{2}f(x_n) + \frac{1}{2}f(x_{n+1})\right) \tag{6.6}$$

となる．

例題として，関数 $f(x, y)$ が y に依存しない $f(x)$ の微分方程式

$$\frac{\mathrm{d}y(x)}{\mathrm{d}x} = f(x) = x^2 - 20x \tag{6.7}$$

を取り上げる．これを積分形にすると

$$y(x) = \int^x t^2 - 20t\,\mathrm{d}t = \frac{1}{3}x^3 - 10x^2 + c \tag{6.8}$$

となり，積分定数 c が 1 個入る．この積分定数は，積分の場合には積分の下限値 (x_0) を指定すれば決まる．また，上限値 (x) も指定すれば，積分は定積分になる．これが微分方程式の解であるときには，初期値 $(x_0, y(x_0))$ を指定することによって c の値が決まる．両者の c の決め方は異なるようにも見えるが，本質的には同じことである．図 6.1 左図の実曲線は式 (6.7) の関数 $f(x)$ であり，右図の実曲線は下限値を $x_0 = 0.0$，あるいは初期値を $(0.0, 0.0)$ とした場合 $(c = 0.0)$ の解の式 (6.8) である．左図で，$h = 1.0$ として台形公式を使って積分すると，右図の関数 $y(x)$ にグラフ上では重なっている値が得られる．右図の一点鎖線のグラフは，下から初期値 $(0.0,\ y(0.0))$ の $y(0.0)$ を 25.0, 50.0, 75.0 としたときのものであり，実曲線を上に平行移動しているだけである．逆に考えると右図の各グラフの勾配が左図の単一の関数 $f(x)$ になるのだから，右図の各 x 点におけるすべての勾配は等しいのである．
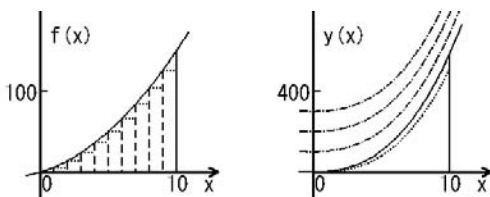
図 6.1　微分方程式 $\dfrac{\mathrm{d}y(x)}{\mathrm{d}x} = f(x, y(x))$ の関数 $f(x)$ (左図) と解の関数 $y(x)$ (右図)．左図の実線は関数 $f(x) = x^2 - 20x$ であり，右図の実線は積分 $y(x) = \dfrac{1}{3}x^3 - 10x^2$ である．左図で，$h = 1.0$ として台形公式を使って微分方程式を解く (積分する) と，右図の関数 $y(x)$ にグラフ上では重なっている値が得られた．右図の一点鎖線のグラフは，下から初期値 $(0.0, y(0.0))$ の $y(0.0)$ を 25.0, 50.0, 75.0 としたときのものである．左図の点線は，微分方程式の解法の粗い近似であるオイラー法の矩形公式を使って式 (6.3) を積分するときの矩形である．得られた解の関数 $y(x)$ が右図に点線で示されている．

他方，関数 $f(x, y)$ が y にも依存している場合は複雑になる．この一般的な場合を論じるために，$y_{n+1} = y(x_n + h)$ を $x = x_n$ でテーラー展開すると，

$$\begin{aligned} y(x_n + h) &= y_n + hy'(x_n) + \frac{1}{2!}h^2 y''(x_n) + \cdots + \frac{1}{n!}h^\ell y^{(\ell)}(x_n) + \cdots \\ &= y_n + hf(x_n, y_n) + \sum_{k=2} \frac{1}{k!}h^k \left[D_x^{k-1} f\right]_n \end{aligned} \tag{6.9}$$

となる．ここで，

$$\begin{aligned} y' &= f \\ y'' &= \frac{\mathrm{d}f(x, y)}{\mathrm{d}x} = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}x} = D_x f \\ y''' &= \frac{\mathrm{d}}{\mathrm{d}x} D_x f = D_x^2 f \\ D_x &\equiv \frac{\partial}{\partial x} + f\frac{\partial}{\partial y} \\ [D_x f]_n &\equiv \left[\left(\frac{\partial}{\partial x} + f\frac{\partial}{\partial y}\right) f\right]_{x_n,\, y_n} \end{aligned} \tag{6.10}$$

を使った．式 (6.9) をどこで打ち切るかという近似の度合いや，次の点の値 y_{n+1} の計算方法によっていろいろな解法が工夫されており，大きく分けると，一段階法，多段階法，予測子・修正子法などがある．

6.2　一段階法

6.2.1　オイラーの前進公式

一番単純な一段階法は式 (6.9) 右辺の第 2 項までを使うオイラー法であり，

$$y_{n+1} = y_n + hf(x_n, y_n) \tag{6.11}$$

となる．これはオイラーの前進公式ともいわれる．使わなかった第 3 項以降は，打ち切り誤差となり，この場合は，

$$\Delta E_{n+1} = y_n + hf(x_n, y_n) - y_{n+1} = -\frac{1}{2}h^2[D_x f]_n + o(h^3) \tag{6.12}$$

なので，h^2 のオーダーである．h を細かくすれば，打ち切り誤差は小さくなるが，計算の回数が増えるので，計算時間がかかるだけでなく，丸めの誤差も蓄積されることになる．図 6.1 左図の点線は，オイラー法の場合の矩形公式を使って積分するための線であり，得られた解の関数 $y(x)$ は右図に点線で示されているように，x が増すに従って正しい解の実線から離れ，誤差が蓄積されていく．右図の一点鎖線のグラフは，下から初期値（0.0, $y(0.0)$）の $y(0.0)$ を 25.0，50.0，75.0 としたときのものであり，グラフは上に平行移動しているだけである．しかし，一般的な微分方程式で，$f(x, y)$ が

y に依存しているなら，右図の各グラフは平行移動しているだけではなく，それなりに異なった形をしていくことになり，同じ x の値でも y の値が異なるとその点での勾配は等しくないことになる．

6.2.2　ルンゲ・クッタ法 2 次公式

オイラー法は式 (6.5) において，$\theta = 0$ とした $y(x_n)$ を使っていることになる．近似を良くするために，まず，オイラー法で $k_1 = f(x_n, y_n)$ を使って $y_{n+1}^{(1)} = y_n + hk_1$ を求める．次にそれを使って $x_{n+1} = x_n + h$ での値 $k_2 = f(x_n + h, y_n + hk_1) = f\left(x_{n+1}, y_{n+1}^{(1)}\right)$ を求め，先の k_1 とともに使って y_{n+1} を決めなおすこともできる．すなわち，

$$\begin{aligned} & y_{n+1}^{(2)} = y_n + h\left(\frac{1}{2}k_1 + \frac{1}{2}k_2\right) \\ & \text{ただし，} k_1 = f(x_n, y_n),\ \ k_2 = f(x_n + h,\ y_n + hk_1) \end{aligned} \tag{6.13}$$

である．$y_{n+1}^{(n)}$ の (n) は近似の程度を区別するものである．k_1 と k_2 の重みを $\dfrac{1}{2}$ とし，k_1 と k_2 の平均値を使うこの公式は，ルンゲ・クッタ法 2 次公式といわれる．k_1 は $x = x_n$ での $y(x)$ の正確な勾配 $f(x_n, y_n)$ であり，k_2 は近似ではあるが $x = x_n + h$ での勾配 $f(x_n + h, y_n + hk_1)$ であるから，この平均を勾配として使うことで，かなり改良されていることになる．次に述べる導出法で調べると，h の 2 次のオーダーまでは正確であり，打ち切り誤差が h^3 のオーダーとなるようになっている．式 (6.13) を拡張して一般的に

$$\begin{aligned} & y_{n+1} = y_n + h\sum_{i=1}^{p} w_i k_i \\ & k_i = f(x_n + \sigma_i h,\ y_n + \sum_{j=1}^{i-1} \sigma_{ij} h k_j) \\ & \text{ただし，} \sigma_i = \sum_{j=1}^{i-1} \sigma_{ij} \end{aligned} \tag{6.14}$$

とする．ここで，$p = 2$ とすると，

$$\begin{aligned} & k_1 = f(x_n, y_n) \\ & k_2 = f(x_n + h\sigma_2,\ y_n + h\sigma_2 k_1) \end{aligned} \tag{6.15}$$

となる．この k_1 と k_2 を (x_n, y_n) の周りで展開した

$$k_2 = f_n + h\sigma_2\left[\frac{\partial f}{\partial x} + f\frac{\partial f}{\partial y}\right]_n + o(h^2) \tag{6.16}$$

を式 (6.14) に代入すると

$$y_{n+1} = y_n + h(w_1 + w_2)f_n + h^2 w_2 \sigma_2\left[\frac{\partial f}{\partial x} + f\frac{\partial f}{\partial y}\right]_n + o(h^3) \tag{6.17}$$

となる．一方，式 (6.9) からの

$$y(x_n + h) = y_n + hf_n + \frac{1}{2!}h^2\left[\frac{\partial f}{\partial x} + f\frac{\partial f}{\partial y}\right]_n + o(h^3) \tag{6.18}$$

と比較すると，$w_1 + w_2 = 1$, $w_2\sigma_2 = \frac{1}{2}$ の関係を得る．ここで，$w_1 = \frac{1}{2}$ を選ぶと式 (6.14) となり，確かに誤差は $o(h^3)$ である．

図 6.2 の左図はルンゲ・クッタ法 2 次公式の説明図であり，右図はルンゲ・クッタ法 4 次公式の説明図である．この場合の微分方程式は y に依存しない $\frac{\mathrm{d}y(x)}{\mathrm{d}x} = \frac{1}{2}x^2$ であり，正確な解 $y(x) = \frac{1}{6}x^3$ のグラフを実線で図に示してある．$x_0 = 0.5$, $x_1 = 1.5$, $h = 1.0$ として，状況をはっきりさせるために刻み h を大きくとってある．点線の矢印は $k_1 = f(x_0)$ であり，オイラー法を使った場合には解の関数の値 $y(x_1)$ はこの点線の矢印の先端であり，正しい解である実線とはかけ離れている．左図の破線の矢印は $k_2 = f(x_1)$ で，実線の矢印は $\frac{k_1 + k_2}{2}$ である．ルンゲ・クッタ法 2 次公式を使った場合には，解の関数の値 $y(x_1)$ は実線の矢印の先端であるから，点線の矢印の先端と比較するとかなり改良されている．ちなみに，式 (6.5) の平均値の定理は $[x_0, x_1]$ の間に正しい解を与える $k = f(\theta h)$ があるという定理であるから，できるだけこの値に近い k を作ればよいということである．関数 $f(x, y)$ が y に依存する場合には，関数 $y(x)$ のグラフの変化は平行移動だけではないので，破線の矢印は $k_2 = f(x_1, y_0 + hk_1)$ となって値も微妙に異なってくる．
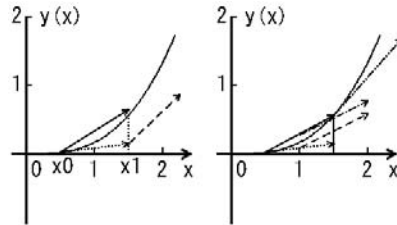
図 6.2　ルンゲ・クッタ法 2 次公式 (左) とルンゲ・クッタ法 4 次公式 (右) の説明図．微分方程式は y に依存しない $\frac{\mathrm{d}y(x)}{\mathrm{d}x} = \frac{1}{2}x^2$ であり，実線の曲線はその正確な解 $y(x) = \frac{1}{6}x^3$ である．左図はルンゲ・クッタ法 2 次公式の場合であり，点線の矢印は $k_1 = f(x_0)$，破線の矢印は $k_2 = f(x_1)$ で，実線の矢印は $\frac{k_1 + k_2}{2}$ である．右図はルンゲ・クッタ法 4 次公式の場合であり，点線の矢印は $k_1 = f(x_0)$，破線の矢印は $k_2 = f\left(\frac{h}{2}, y_0 + \frac{h}{2}k_1\right)$，一点鎖線の矢印は $k_3 = f\left(\frac{h}{2}, y_0 + \frac{h}{2}k_2\right)$，二点鎖線の矢印は $k_4 = f(x_1, y_0 + hk_3)$ で，実線の矢印は $k = \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$ である．

6.2.3　ルンゲ・クッタ法 4 次公式

ルンゲ・クッタ法 2 次公式では式 (6.14) で $p = 2$ としているが，$p = 4$ とした公式はルンゲ・クッタ法 4 次公式といわれ，最もよく使われている．その場合には，

$$\begin{aligned} &y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\ &k_1 = f(x_n,\, y_n), \quad k_2 = f\left(x_n + \frac{h}{2},\, y_n + \frac{h}{2}k_1\right) \\ &k_3 = f\left(x_n + \frac{h}{2},\, y_n + \frac{h}{2}k_2\right), \quad k_4 = f(x_n + h,\, y_n + hk_3) \end{aligned} \tag{6.19}$$

である．k_1 は $x = x_n$，k_2 と k_3 は $x = x_n + \dfrac{h}{2}$，k_4 は $x = x_n + h$ での値であり，図 6.2 右図に示されている．実際に使う k としては，式 (6.19) の第 1 式からわかるように，x_n，$x_n + \dfrac{h}{2}$，$x_n + h$ での重みが $\dfrac{1}{6}$，$\dfrac{4}{6}$，$\dfrac{1}{6}$ となっているから，シンプソンの積分公式に類似したものである．この公式の打ち切り誤差は h^5 のオーダーである．

6.3　多段階法と予測子・修正子法

式 (6.3) を等間隔にとった分点を使って求める場合には，式 (6.4) の代わりに少し前 (x_{n-q}) の値にまでさかのぼって，

$$y_{n+1} = y_{n-q} + \int_{x_{n-q}}^{x_n+h} f(x,\, y(x))\mathrm{d}x \tag{6.20}$$

を使うことができる．ただし，$n \geq q$ でなければならない．また，$0 \leq i < q$ の y_i については，別の方法で求めておかなければならない．$q > 0$ のとき，この方法は多段階法といわれる．既知の y_i を使う補間法を適応すると定積分は解析的に求められ，式 (6.20) は

$$\begin{aligned} y_{n+1} &= y_{n-q} + h\sum_{i=0}^{p} w_i f(x_{n-i},\, y_{n-i}) \\ &= y_{n-q} + h\sum_{i=0}^{p} w_i f_{n-i} \end{aligned} \tag{6.21}$$

となる．q と p の各組み合わせについて，w_i の値を決めることができる．この方法は，すべて既知の f の値を使っているので，陽的な多段階法といわれる．

一方，式 (6.21) において，未知の f_{n+1} まで使う方法は陰的な多段階法といわれ，

$$\begin{aligned} y_{n+1} &= y_{n-q} + h\sum_{i=0}^{p} w_i f(x_{n+1-i},\, y_{n+1-i}) \\ &= y_{n-q} + h\sum_{i=0}^{p} w_i f_{n+1-i} \end{aligned} \tag{6.22}$$

である．この方法を使うためには，ルンゲ・クッタ法の場合のように，前もって f_{n+1} の値の予測値を求めなければならない．q と p の各組み合わせについて，w_i の値が定められており，例えば

$$\begin{aligned} & q=0,\ p=1: \\ & \quad y_{n+1}=y_n+\frac{h}{2}(f_n+f_{n+1}) \qquad \text{台形公式} \\ & q=1,\ p=2: \\ & \quad y_{n+1}=y_{n-1}+\frac{h}{3}(f_{n-1}+4f_n+f_{n+1}) \quad \text{シンプソンの公式} \end{aligned} \tag{6.23}$$

がある．このような方法は予測子・修正子法といわれ，陽的な多段階法 (予測子) と陰的な多段階法 (修正子) の組み合わせで，いろいろな方法がある．例えば，

❐ ミルン法

$$\begin{aligned} &(\text{予測子}) \qquad y_{n+1}=y_{n-3}+\frac{4}{3}h(2f_{n-2}-f_{n-1}+2f_n) \\ &(\text{修正子}) \qquad y_{n+1}=y_{n-1}+\frac{h}{3}(f_{n-1}+4f_n+f_{n+1}) \end{aligned} \tag{6.24}$$

❐ アダムス・ムルトン法

$$\begin{aligned} &(\text{予測子}) \qquad y_{n+1}=y_n+\frac{h}{24}(-9f_{n-3}+37f_{n-2}-59f_{n-1}+55f_n) \\ &(\text{修正子}) \qquad y_{n+1}=y_n+\frac{h}{24}(f_{n-2}-5f_{n-1}+19f_n+9f_{n+1}) \end{aligned} \tag{6.25}$$

などである．

6.4 高階常微分方程式（人工衛星の軌道計算）

高階常微分方程式

$$\frac{\mathrm{d}^m y}{\mathrm{d}x^m}=f(x,\,y,\,y',\,\cdots,\,y^{(m-1)}) \tag{6.26}$$

は，例えば，y を y_1 に書き換えるなどをして，

$$\left.\begin{aligned} \left(\frac{\mathrm{d}y}{\mathrm{d}x}=\right) &\quad y_1'=y_2 \\ \left(\frac{\mathrm{d}^2y}{\mathrm{d}x^2}=y''=\right) &\quad y_2'=y_3 \\ &\quad \cdots \\ \left(\frac{\mathrm{d}^m y}{\mathrm{d}x^m}=\right) &\quad y_m'=f(x,\,y_1,\,y_2,\,\cdots,\,y_m) \end{aligned}\right\} \tag{6.27}$$

のような m 元連立 1 階常微分方程式とすることができる．各階の方程式の選び方は任意であるから，これ以外に適当に選ぶこともできる．実際，式 (6.35) はそうである．次に，連立 1 階常微分方程式の解説を行う．

6.4.1　連立1階常微分方程式

m 個の従属変数をもつ m 元連立1階常微分方程式の場合には，解をベクトル関数 $\mathbf{y} = (y_1, \cdots, y_m)$ で表す．連立方程式と初期条件は

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}x} = \mathbf{f}(x, \mathbf{y}), \qquad x = x_0 \quad \text{で} \quad \mathbf{y} = \mathbf{y}(x_0) \tag{6.28}$$

である．これに，ルンゲ・クッタ法4次公式を適用すると，

$$\begin{aligned}
&\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4) \\
&\mathbf{k}_1 = \mathbf{f}(x_n, \mathbf{y}_n), \quad \mathbf{k}_2 = \mathbf{f}\left(x_n + \frac{h}{2}, \mathbf{y}_n + \frac{h}{2}\mathbf{k}_1\right) \\
&\mathbf{k}_3 = \mathbf{f}\left(x_n + \frac{h}{2}, \mathbf{y}_n + \frac{h}{2}\mathbf{k}_2\right), \quad \mathbf{k}_4 = \mathbf{f}(x_n + h, \mathbf{y}_n + h\mathbf{k}_3)
\end{aligned} \tag{6.29}$$

である．

プログラム (RungeKutta4.java) のメソッド (rungeKutta4) と，その説明は以下のとおりである．

RungeKutta4.java のメソッド (rungeKutta4)：

```
 1  public void rungeKutta4(double[] y, MyFunction1 f,
 2        double h, int isn) {
 3    int ny= y.length;
 4    double[] yw= new double[ny];
 5    double[] k1= new double[ny];
 6    double[] k2= new double[ny];
 7    double[] k3= new double[ny];
 8    double[] k4= new double[ny];
 9    double h2= h/2;
10    for (int is=0; is<isn; is++) {
11      k1= f.function(y);
12
13      for (int i=0; i<ny; i++) yw[i]= y[i]+h2*k1[i];
14      k2= f.function(yw);
15
16      for (int i=0; i<ny; i++) yw[i]= y[i]+h2*k2[i];
17      k3= f.function(yw);
18
19      for (int i=0; i<ny; i++) yw[i]= y[i]+h*k3[i];
20      k4= f.function(yw);
21
22      y[0]= y[0]+h;
23      for (int i=1; i<ny; i++) {
24        y[i]= y[i]+h6*(k1[i]+2*k2[i]+2*k3[i]+k4[i]);
25      }
26    }
27  }
```

1. L1, 2〜27: メソッド rungeKutta4 の定義である．引数の y は配列で，y[0] は微分方程式 $\mathbf{y}' = \mathbf{f}(x, \mathbf{y})$ の独立変数 x，　y[1] 以降は，ベクトル関数 $\mathbf{y}(x)$ の配列である．メソッドから戻るときには，isn 刻み進んだ変数値と関数値が代入されている．f はインタフェース MyFunction1 クラスのインスタンスであり，解を求めるメソッドが定義されている．f のメソッドには連立微分方程式が定義されており，戻り値は $\mathbf{f}(x, \mathbf{y})$ の配列である．h は独立変数 x の刻み幅で，isn は isn 回刻みを進んだらこのメソッドから戻ることを指定している．
2. L4: 関数 $\mathbf{f}(x, \mathbf{y})$ の $x, \mathbf{y}$ のための作業用配列である．
3. L5〜8: 式 (6.29) の $\mathbf{k}_1$〜$\mathbf{k}_4$ の配列である．
4. L9: $\dfrac{h}{2}$ を定義する．
5. L10〜26: 指定された回数だけ，刻みを進める．
6. L11: 式 (6.29) の $\mathbf{k}_1$ の計算．
7. L13, 14: 式 (6.29) の $\mathbf{k}_2$ の計算．
8. L16, 17: 式 (6.29) の $\mathbf{k}_3$ の計算．
9. L19, 20: 式 (6.29) の $\mathbf{k}_4$ の計算．
10. L22〜24: 式 (6.29) の $\mathbf{y}_{n+1}$ の計算．

6.4.2　人工衛星，惑星の軌道計算（ルンゲ・クッタ法 4 次公式）

人工衛星や惑星は，地球あるいは太陽の引力を受けて，それぞれの周りを楕円軌道を描いて周遊している．引力は万有引力であり，その強さは引力の中心からの距離の 2 乗に反比例している．そこで，運動方程式は

$$m\frac{\mathrm{d}^2\mathbf{r}(t)}{\mathrm{d}t^2} = -G\frac{mM}{r^2(t)}\hat{\mathbf{r}}(t) \tag{6.30}$$

である．ここで，m は衛星あるいは惑星の質量，G は万有引力定数，M は地球あるいは太陽の質量，$\mathbf{r}(t)$ は衛星あるいは惑星の位置ベクトル，$\hat{\mathbf{r}}(t)$ は位置ベクトルの単位ベクトル，t は時間である．実際には，3 次元空間での運動であるが，1 個の衛星あるいは惑星を問題にするのであれば，楕円軌道面を含む 2 次元平面に限定することができる．そこで，ベクトル $\mathbf{r}$ を位置座標 (x, y) とする．運動方程式を位置座標 (x, y) を使って，微分方程式に書きなおすと

$$\left.\begin{aligned}\frac{\mathrm{d}^2x}{\mathrm{d}t^2} &= -GM\frac{x}{(x^2+y^2)^{\frac{3}{2}}}\\ \frac{\mathrm{d}^2y}{\mathrm{d}t^2} &= -GM\frac{y}{(x^2+y^2)^{\frac{3}{2}}}\end{aligned}\right\} \tag{6.31}$$

である．これは，2 元連立 2 階常微分方程式であるから，これを 4 元連立 1 階常微分方程式

$$\left.\begin{array}{rcl}\dfrac{\mathrm{d}x}{\mathrm{d}t} & = & v_x \\ \dfrac{\mathrm{d}v_x}{\mathrm{d}t} & = & -GM\dfrac{x}{(x^2+y^2)^{\frac{3}{2}}} \\ \dfrac{\mathrm{d}y}{\mathrm{d}t} & = & v_y \\ \dfrac{\mathrm{d}v_y}{\mathrm{d}t} & = & -GM\dfrac{y}{(x^2+y^2)^{\frac{3}{2}}}\end{array}\right\} \tag{6.32}$$

に書きなおす．ここまで来れば，ベクトル関数に対するルンゲ・クッタ法 4 次公式を適用することができる．

図 6.3 は，ルンゲ・クッタ法 4 次公式を使って計算した人工衛星の軌道である．初期条件として，特定の地点の上空 y km と水平方向の速度 v_x km · s^{-1} の 4 組 (0.0, 8.0)，(0.0, 9.0)，(0.0, 10.0)，(0.0, 11.0) を与えている．ただし，計算に使う y の初期値としては，初期条件として与える地表からの高さに地球の半径を加えている．

プログラム (RungeKutta4Test.java) の連立方程式のクラス (Func) と，その説明は以下のとおりである．
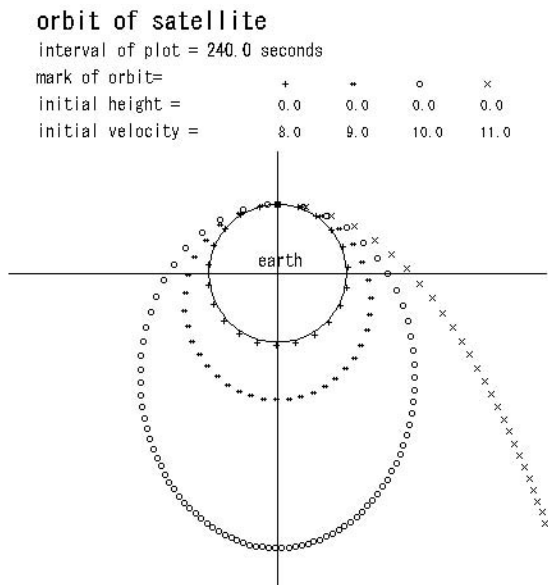
図 6.3　ルンゲ・クッタ法 4 次公式を使った人工衛星の軌道計算．初期条件として，特定の地点の上空 y km で水平方向に速度 v_x km · s^{-1} で打ち出す y と v_x を与える．（Z06_03_orbit_pro.java）

クラス (Func) のメソッド (function)：

```
 1 class Func implements MyFunction1 {
 2   public double[] function(double[] y) {
 3     double gr= 6.67e-11;    //  万有引力定数
 4     double rmass= 5.977e+24;//  地球の質量
 5     double GM= gr*rmass ;
 6     double[] f= new double[y.length];
 7     double r32= Math.pow(y[1]*y[1]+y[3]*y[3],-1.5);
 8     f[0]= 1.0;
 9     f[1]= y[2];
10     f[2]= -GM*y[1]*r32;
11     f[3]= y[4];
12     f[4]= -GM*y[3]*r32;
13     return f;
14   }
15 }
```

1. L1〜15: インタフェース MyFunction1 をインプリメント (実装) するクラス Func の定義である．
2. L2: メソッド function のオーバーライドである．引数 y は配列で，y[0] は微分方程式 $\mathbf{y}' = \mathbf{f}(x, \mathbf{y})$ の独立変数 x， y[1] 以降は，ベクトル関数 $\mathbf{y}(x)$ の配列である．戻り値は式 (6.28) の $\mathbf{f}$ の配列である．
3. L3〜5: 万有引力定数や地球の質量などの定数である．
4. L6: 戻り値のための配列である．
5. L7: 式 (6.32) の距離の 3 乗の逆数である．
6. L8: 独立変数自身の微分であるから，常に 1 である．
7. L9〜12: 式 (6.32) の計算である．
8. L13: 戻り値 f を返す．

6.5 ルンゲ・クッタ・ジル法 (水素原子の電子状態)

微分方程式の解をある範囲にわたってルンゲ・クッタ法で求めるとき，刻み h を大きくすると打ち切り誤差は h^5 のオーダーで大きくなるから，刻みを小さくする必要がある．そこで，ある範囲での解を求めるために，数百から数千回繰り返して計算をする必要があることもある．それを少しでも改善するために，丸めの誤差ができるだけ少なくなるように，ジルによって改良された，ルンゲ・クッタ・ジル法といわれるアルゴリズムがある．それは，式 (6.19) の k_i の計算に付加項を付けるものであり，

$$
\begin{aligned}
&q_0 = \begin{cases} 0 & 1\text{ 回目} \\ q_4 & 2\text{ 回目以降は直前の } q_4 \text{を使う} \end{cases} \\
k_1 = \ & hf(x_0,\ y_0) \\
&y_1 = y_0 + \frac{1}{2}(k_1 - 2q_0) \\
&q_1 = q_0 + 3\left[\frac{1}{2}(k_1 - 2q_0)\right] - \frac{1}{2}k_1 \\
k_2 = \ & hf\left(x_0 + \frac{h}{2},\ y_1\right) \\
&y_2 = y_1 + \left(1 - \sqrt{\frac{1}{2}}\right)(k_2 - q_1) \\
&q_2 = q_1 + 3\left[\left(1 - \sqrt{\frac{1}{2}}\right)(k_2 - q_1)\right] - \left(1 - \sqrt{\frac{1}{2}}\right)k_2 \\
k_3 = \ & hf\left(x_0 + \frac{h}{2},\ y_2\right) \\
&y_3 = y_2 + \left(1 + \sqrt{\frac{1}{2}}\right)(k_3 - q_2) \\
&q_3 = q_2 + 3\left[\left(1 + \sqrt{\frac{1}{2}}\right)(k_3 - q_2)\right] - \left(1 + \sqrt{\frac{1}{2}}\right)k_3 \\
k_4 = \ & hf(x_0 + h,\ y_3) \\
&y_4 = y_3 + \frac{1}{6}(k_4 - 2q_3) \\
&q_4 = q_3 + 3\left[\frac{1}{6}(k_4 - 2q_3)\right] - \frac{1}{2}k_4
\end{aligned}
\tag{6.33}
$$

である．

図 6.4 はルンゲ・クッタ・ジル法による水素原子における電子の $1s$ 動径波動関数である．この場合の微分方程式は動径シュレーディンガー方程式といわれ

$$
-\frac{\mathrm{d}^2 P_l(r)}{\mathrm{d}r^2} + \left\{-\frac{2}{r} + \frac{l(l+1)}{r^2}\right\} P_l(r) = EP_l(r) \tag{6.34}
$$
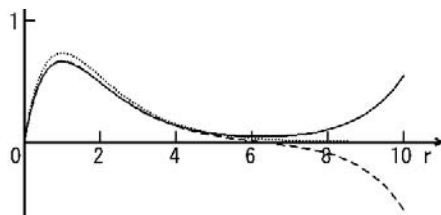
図 6.4　ルンゲ・クッタ・ジル法による水素原子動径波動関数．点線が，固有エネルギーが $E = -1\mathrm{Ry}$ で境界条件を満たす解 (固有解) である．実線と鎖線はエネルギーが $\Delta E = -0.001, 0.001$ ずつずれている場合であり，無限遠で発散しており固有解ではない (注：計算誤差があるために，この図の計算では $E = -1\mathrm{Ry}$ ではなく，$E = -0.9990\mathrm{Ry}$ であった)．(Z06_04_Rkg_H.java)

である．この式は，エネルギーの単位を Ry とした原子単位系で書かれている．位置座標 r は，電子の原子核からの距離であり，$P_l(r)$ は電子の動径波動関数である．l は電子の方位量子数といわれ $l \geq 0$ であり，計算に先立ち指定しておく．E は電子のエネルギー準位であり，$-\dfrac{2}{r}$ は位置エネルギーである．解の境界条件は，$P_l(0) = 0$, $P_l'(0) \propto l$，$P_l(\infty) = 0$ である．もし，解の境界条件が初期条件の $P_l(0) = 0$，$P_l'(0) \propto l$ だけであれば，あらゆる E について解となる動径関数を求めることができるが，電子が原子に束縛されており，遠くにある確率はゼロであるという物理的な条件，$P_l(\infty) = 0$ がもう一つの境界条件となり，この条件を満たす解のみが物理的に意味のある解となる．

式 (6.34) を連立 1 階微分方程式に変形するために，$Q(r) = P'(r) - \dfrac{l+1}{r}P(r)$ と置くと

$$\left.\begin{array}{rcl} P'(r) & = & Q(r) + \dfrac{l+1}{r}P(r) \\ Q'(r) & = & \left(-\dfrac{2}{r} + E\right)P(r) - \dfrac{l+1}{r}Q(r) \end{array}\right\} \tag{6.35}$$

となる．

エネルギー固有値が最も低い $l = 0$ の 1s 状態のエネルギーは $E_{1s} = -1.0$ なので，その近傍を検討する．図 6.4 の点線の動径波動関数 $P_{1s}(r)$ が 1s 状態に対応するもので，無限遠でちょうどゼロに収束している．

プログラム (Z06_04_Rkg_H.java) の一部と，その説明は以下のとおりである．

Z06_04_Rkg_H.java の一部：

```
  1 //  図6.4 水素の動径波動関数 Z06_04_Rkg_H.java 04/12/04

 49 class Z06_04_Rkg_HGraph {

 57   public void mydraw() {

 85     int lmax= 0;
 87     double z= 1;

 89     double[] xdivp= {0.0, 0.01, 0.05, 0.25, 1.0, 2.5, 5.0,10.0};
 90     int[] numbdx= {20, 20, 20, 20, 20, 20, 20};

 99     int ntotv= ntot+1;
101     int npott=2*ntot+1;
102     double[] rpot= new double[npott];
103     double[] pot= new double[npott];

105     double eamin=-0.99999;
106     double eamax=-0.99798;
107
108     int ne= 3;
109     double[] e= new double[ne];
```

```
117     double[] xrkg= new double[ntotv];
118     double[][] yrkg= new double[ntotv][ne];

121     int n= 2;
122     xdivp[0]= (xdivp[1]-xdivp[0])/numbdx[0];
123     double x= xdivp[0];
124     numbdx[0]= numbdx[0]-1;
125     pot[n]= -2*z/x;

138     Function func= new Function();
139     for (l=0; l<=lmax; l++) {
140       logd(l, e, pot, z, xdivp, numbdx, func, xrkg, yrkg);

146     }

177   }

179   public void logd(int l, double[] e, double[] pot,
180     double z, double[] xdivp, int[] numbdx,
181     Function func, double[] xrkg, double[][] yrkg) {

185     double[][] yw= new double[2][ne];

188     double fl1= l+1;
189     px0= xdivp[0];
190     vx0= pot[2];
191     for (int i=0; i<ne; i++) {

197     }

199     Rkg_atom rk= new Rkg_atom();
200     rk.rkg_atom(l, xdivp, numbdx, func, yw, xrkg, yrkg, pot, e);
201     return;
202   }
203 }

205 class Rkg_atom {
206     public void rkg_atom(int l, double[] xdivp, int[] numbdx,
207     Function funct, double[][] yw, double[] x, double[][] y,
208     double[] pot, double[] e) {

315 }

318 class Function{
319   public void function(int l, double x, double[][] yw, double[][] ypw,
320     int ipn, double[] pot, double[] e) {
321
322     int ne= e.length;
323     double vrr=pot[ipn];
324     double r2=x*x;
```

```
325      double fl1= l+1;
326
327      for (int i=0; i<ne; i++) {
328        ypw[0][i]=yw[1][i]+fl1*yw[0][i]/x;
329        ypw[1][i]=(vrr-e[i])*yw[0][i]-fl1*yw[1][i]/x;
330      }
331      return;
332    }
333 }
```

1. L49〜203: 計算と描画をするクラス Z06_04_Rkg_HGraph の定義である．
2. L57〜177: 計算と描画をするメソッド mydraw の定義である．
3. L85, 87: lmax は計算する方位量子数の最大値であり，ここでは 1s 軌道だけであるから 0 である．z は原子番号であり，ここでは水素であるから 1 である．
4. L89, 90: ルンゲ・クッタ・ジル法は x 座標の等分割を使うから，関数の変化の激しい原点近傍から遠方に行くに従って，刻み幅を段階的に広くしている．xdivp は刻み幅を変える x 軸の分点であり，numbdx はその間の等分割点の数である．
5. L99〜103: ntot は分割点の総数，ntotv は求める波動関数の点の数，npott は用意しておくポテンシャルのデータ総数である．ルンゲ・クッタ・ジル法の 1 刻みで，ポテンシャルは 2 点で必要であるから，npott は 2 倍されている．rpot は座標であり， pot はポテンシャル値である．
6. L105〜109: 波動関数を求めるエネルギー値 e，数 ne，最小値 eamin，最大値 eamax である．
7. L117, 118: xrkg は分点の座標，yrkg は答えの波動関数である．
8. L121〜124: ポテンシャルは原点で発散しているから，原点を初期値とすることはできないので，1 刻み進んだ点 (n=2) を初期値にする．そのための変更である．
9. L125: 初期条件でのポテンシャルの値を決めている．以後，各分点でのポテンシャルの値を pot に代入している．
10. L138: 微分方程式のクラス (L318〜333) のインスタンスを作成している．
11. L139〜146: 各方位量子数ごとにすべてのエネルギーに対する解を求める．
12. L179〜203: このメソッド logd では微分方程式を解く準備をし，微分方程式を解き，解いた後の処理をする．
13. L185: ルンゲ・クッタ・ジル法の各刻みでの解を代入する作業用配列である．
14. L188〜197: 波動関数の初期値を決める際，微分方程式を級数展開して 2 次の項までとり，精度を上げている．
15. L199, 200: 原子の波動関数を求めるルンゲ・クッタ・ジル法 Rkg_atom のインスタンスを作り波動関数を求めている．
16. L205〜315: ルンゲ・クッタ・ジル法のクラス Rkg_atom である．刻み幅を変えられるようになっている．
17. L318〜333: 2 元連立方程式 (6.35) が定義されているクラス Funtion である．

図 6.5 は，ルンゲ・クッタ・ジル法により求められた，水素原子における電子の動径波動関数の直交規格性を示す図である．下図の実線と破線は 1s と 2s の動径波動関数であり，固有エネルギーは 1s が $E = -1\text{Ry}$，2s が $E = -1/4\text{Ry}$ である．点線は 1s
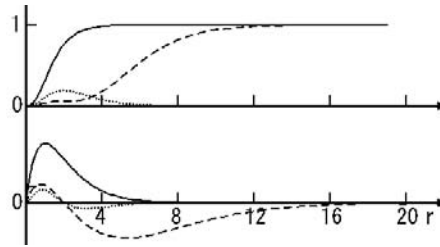
図 6.5　水素原子動径波動関数の直交規格性．下図の実線と破線は 1s と 2s の動径波動関数であり，点線は 1s と 2s の動径波動関数の積である．上図の実線と破線は 1s と 2s の関数の 2 乗を 0 から r まで積分した値であり，点線は 1s と 2s の関数の積を 0 から r まで積分した値である．（Z06_05_Rkg_H_1s2s.java）

と 2s の動径波動関数の積である．上図の実線と破線は 1s と 2s の関数の 2 乗を 0 から r まで積分した値 $\int_0^r |P_{ns}(r')|^2 \mathrm{d}r'$ であり，r が大きくなると 1 に収束して，関数が規格化されていることを示している．点線は 1s と 2s の関数の積を 0 から r まで積分した値 $\int_0^r P_{1\mathrm{s}}(r')P_{2\mathrm{s}}(r')\mathrm{d}r'$ であり，r が大きくなると 0 に収束して，両関数が直交していることを示している．

演習問題

6.1　速度に比例する抵抗力を受ける単振動は減衰振動となり，運動方程式は $\dfrac{\mathrm{d}x^2}{\mathrm{d}t^2} = -2k\dfrac{\mathrm{d}x}{\mathrm{d}t} - \omega^2 x$ である．ここで，$2k\dfrac{\mathrm{d}x}{\mathrm{d}t}$ は抵抗力，$\omega^2 x$ は復元力である．$\omega = 1$ とし，$k = 0.0$, 0.1, 3.0 の場合に，どのような振動をするか調べなさい．ただし，初期条件は $t = 0$ で $x = 1$, $\dfrac{\mathrm{d}x}{\mathrm{d}t} = 0$ とする．

解答例： ソースプログラム Q06_01_Z06_03_shindou.java

第 7 章

連立１次方程式の解法

連立 1 次方程式は数値解析の多くの場面に現れ，本書でも，最小二乗法，固有値問題の直接法，スプライン関数を扱うときなどで，陰に陽に使われている．連立方程式などを行列形式で表現すると極めて簡潔になるのと同じように，最近は，行列計算を行うハードウェアやプログラミング言語も存在している．しかし，それらを利用することは非常に高価であり，なかなか使えるものではないから，これまでどおり，ループをまわして計算をする必要がある．

単純にいうと，$(n \times n)$ 型の行列の大きさ n が 2 倍になると，処理は 8 倍になるので，行列の大きさに応じてアルゴリズムの使い分けが必要であり，また，ゼロとなっている要素が少ない密行列と，それが多い疎行列とでも，アルゴリズムの使い分けが必要である．

なお，本章の本文では行列要素などを $a_{11} \sim a_{nn}$ のように $1 \sim n$ としているが，Java プログラムでは a[0][0] 〜 a[n−1][n−1] のように添字番号を $0 \sim n-1$ としていることに注意してほしい．

7.1 連立１次方程式

まず，次の 3 元連立 1 次方程式

$$\left.\begin{array}{rrrrl} 3x- & y+ & 2z= & 1 & \cdots(1)\\ x+ & y+ & z= & 0 & \cdots(2)\\ 2x- & 4y+ & 3z= & -1 & \cdots(3) \end{array}\right\} \tag{7.1}$$

を解いてみよう．x を消去すると

$$\left.\begin{array}{rrrrrl} & 3x\ -\ y+ & 2z= & 1 & \cdots(1)' \\ 3*(2)-(1): & 4y+ & z= & -1 & \cdots(2)' \\ (3*(3)-2*(1))/5: & -2y+ & z= & -1 & \cdots(3)' \end{array}\right\} \tag{7.2}$$

となる．$(3)'$ では簡単にするために 5 で割ってある．次に y を消去すると，

$$\left.\begin{array}{rrrrl} & 3x\ -\ y+ & 2z= & 1 & \cdots(1)'' \\ & 4y+ & z= & -1 & \cdots(2)'' \\ 2*(3)'+(2)': & & 3z= & -3 & \cdots(3)'' \end{array}\right\} \tag{7.3}$$

となる．(3)″ から順に z, y, x を求めていくと，解

$$z = -1, \quad y = 0, \quad x = 1 \tag{7.4}$$

が定まる．

7.1.1　ガウスの消去法

計算機のプログラミングにおけるガウスの消去法は，式 (7.1) 〜式 (7.4) の方法と原理的には同じ方法である．ガウスの消去法において，式 (7.1) 〜式 (7.3) の消去の過程は前進消去といわれ，式 (7.4) で z から順に答えを求めていく過程は後退代入といわれる．これからは n 元連立 1 次方程式

$$\left.\begin{array}{c} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \cdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{array}\right\} \tag{7.5}$$

を扱う．この方程式を行列を用いて表すと

$$A\mathbf{x} = \mathbf{b} \tag{7.6}$$

である．ここで，

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ & & \cdots & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \cdot \\ x_n \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \cdot \\ b_n \end{pmatrix} \tag{7.7}$$

であり，A は係数行列といわれる．式 (7.1) 〜式 (7.3) へと進むに従って，′ なし，′ 付き，″ 付きで段階を区別したが，これからは，式 (7.5) を第 0 段階として，上付き添字 (0), (1), $\cdots$ を付けることにする．以後，本文の説明では，ベクトル $\mathbf{b}$ を行列 A の第 $n+1$ 列ベクトルとして，A を n 行 $n+1$ 列の行列に拡張し，$\mathbf{b}$ も同じ式で扱えるようにする．そこで，前進消去の第 1 段階の操作は，第 1 行についての

$$a_{1j}^{(1)} = a_{1j}^{(0)} / a_{11}^{(0)} \qquad (\text{列 } j = 1,\ 2,\ \cdots,\ n+1) \tag{7.8}$$

と，行 $i = 2, 3, \cdots, n$ についての

$$a_{ij}^{(1)} = a_{ij}^{(0)} - a_{i1}^{(0)} a_{1j}^{(1)} \qquad (\text{列 } j = 1,\ 2,\ \cdots,\ n+1) \tag{7.9}$$

である．式 (7.2) を導き出すときと異なり，1 行目で $a_{11}^{(1)} = 1$ となるように変更し，2 行目以下もプログラミングに都合が良いように変えてある．この操作の結果 A は

$$A^{(1)} = \begin{pmatrix} 1.0 & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} & a_{1n+1}^{(1)} \\ 0.0 & a_{22}^{(1)} & \cdots & a_{2n}^{(1)} & a_{2n+1}^{(1)} \\ & & \cdots & & \\ 0.0 & a_{n2}^{(1)} & \cdots & a_{nn}^{(1)} & a_{nn+1}^{(1)} \end{pmatrix} \tag{7.10}$$

となる．$1 \leq k \leq n-1$ として，第 k 段階の操作は，行 k についての

$$a_{kj}^{(k)} = a_{kj}^{(k-1)}/a_{kk}^{(k-1)} \qquad (\text{列 } j = k, \cdots, n+1) \tag{7.11}$$

と，行 $i = k+1, \cdots, n$ についての

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - a_{ik}^{(k-1)} a_{kj}^{(k)} \qquad (\text{列 } j = k, \cdots, n+1) \tag{7.12}$$

である．最後の第 n 段階の操作は

$$a_{nj}^{(n)} = a_{nj}^{(n-1)}/a_{nn}^{(n-1)} \qquad (\text{列 } j = n, n+1) \tag{7.13}$$

だけである．

第 k 段階の操作を行列で表現すると

$$O^{(k)} = \begin{pmatrix} 1.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 \\ 0.0 & 0.0 & \dfrac{1}{a_{kk}^{(k-1)}} & 0.0 & \cdots & 0.0 & 0.0 \\ 0.0 & 0.0 & -\dfrac{a_{(k+1)k}^{(k-1)}}{a_{kk}^{(k-1)}} & 1.0 & \cdots & 0.0 & 0.0 \\ 0.0 & 0.0 & -\dfrac{a_{(k+2)k}^{(k-1)}}{a_{kk}^{(k-1)}} & 0.0 & 1.0 & \cdots & 0.0 \\ . & . & . & . & \cdots & . & . \\ 0.0 & 0.0 & -\dfrac{a_{nk}^{(k-1)}}{a_{kk}^{(k-1)}} & 0.0 & \cdots & 0.0 & 1.0 \end{pmatrix} \tag{7.14}$$

を使って

$$A^{(k)} = O^{(k)} A^{(k-1)} \tag{7.15}$$

と表せるが，この行列の積をまともに行うと無駄が多く，処理時間がかかるので，プログラムでは式 (7.11) と式 (7.12) に定義されている変換操作のみを行う．すべての操作の結果 A は上三角行列

$$A^{(n)} = \prod_{k=1}^{n} O^{(k)} A = OA = \begin{pmatrix} 1.0 & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} & a_{1n+1}^{(1)} \\ 0.0 & 1.0 & \cdots & a_{2n}^{(2)} & a_{2n+1}^{(2)} \\ & & \cdots & & \\ 0.0 & 0.0 & \cdots & 1.0 & a_{nn+1}^{(n)} \end{pmatrix} \tag{7.16}$$

となる．解を求めるための後退代入は，

$$\begin{aligned} x_n &= a_{nn+1}^{(n)} \\ x_i &= a_{in+1}^{(n)} - \sum_{j=i+1}^{n} a_{ij}^{(i)} x_j \qquad (\text{行 } i = n-1, \cdots, 1) \end{aligned} \tag{7.17}$$

である．第 k 段階の式 (7.11) で $a_{kk}^{(k-1)}$（ピボット）で割る操作があるが，それが 0 である場合には割り算ができない．そのようなことを避けるためにばかりでなく，桁落ち

で精度が下がらないようにするためにも，第 k 段階で $a_{ik}^{(k-1)}$（行 $i = k, \cdots, n$）の中で絶対値が最大の項が第 k 行に来るように行の入れ替えを行う．行の入れ替えは，連立方程式の式の順序を入れ替えることであるから，結果には影響を与えない．一方，列に関しても絶対値が最大の項が第 k 列に来るように列の入れ替えを行うこともできる．この場合は解 x_k の順番に入れ替えが起こるから，その場合には入れ替えを記録しておき，最後に元に戻さなければならない．このように，絶対値が最大な項をピボットとして使うことをピボットの選択という．

プログラム (Gauss_Shoukyohou.java) のメソッド (gauss_Shoukyohou) と，その説明は以下のとおりである．ピボット選択は行の入れ替えだけで行っている．

Gauss_Shoukyohou.java のメソッド (gauss_Shoukyohou)：

```
 1     public double[] gauss_Shoukyohou(int n, double[][] a,
 2         double[] b)  {
 3     double[] x= new double[n];
 4     double w;
 5     int ip= 0;
 6     for (int k=0; k<n; k++) {
 7       w= 0.0;
 8       for (int i=k; i<n; i++) {
 9         if (Math.abs(a[i][k])  >  w) {
10           w = Math.abs(a[i][k]);
11           ip = i;
12         }
13       }
14       if (ip != k) {
15         for (int j=k; j<n; j++) {
16           w= a[k][j];
17           a[k][j]= a[ip][j];
18           a[ip][j]= w;
19         }
20         w= b[k];
21         b[k]= b[ip];
22         b[ip]= w;
23       }
24 //  前進消去開始
25       double akk= 1.0/a[k][k];
26       double aik;
27       for (int j=k+1; j<n; j++) {
28         a[k][j]= a[k][j]*akk;
29       }
30       b[k]= b[k]*akk;
31       for (int i=k+1; i<n; i++) {
32         aik= a[i][k];
33         for (int j=k+1; j<n; j++) {
34           a[i][j] -= aik*a[k][j];
35         }
36         b[i] -= aik*b[k];
37       }
```

```
38      }
39      for (int i=n-1; i>=0; i--) {
40        for ( int j= i+1; j<n; j++) {
41          b[i] -= a[i][j]*b[j];
42        }
43        x[i] = b[i];
44      }
45      return x;
46    }
```

1. L1, 2〜46: メソッド gauss_Shoukyohou の定義である．連立方程式は $a_{ij} * x_j = b_i$ で，戻り値は解のベクトル x である．引数の n は係数行列 a の次元，a は係数行列，b は右辺の定数ベクトルである．
2. L3: 解のベクトル x の配列である．
3. L4, 5: ピボットの選択をするときに使う作業用の変数である．
4. L6〜38: 第 k 段階の前進消去を行うループである．
5. L7〜23: ピボットの選択を行う．
6. L8〜13: 対象となる第 k 列の対角要素から下で絶対値が最大な要素を選び，その行番号を ip に代入する．
7. L14: ピボットが第 k 行かどうか調べる．
8. L15〜22: ピボットが第 k 行でなかったら，第 k 行と最大の要素がある行との入れ替えを行う．プログラムでは a[k][n+1] を b[k] としている．
9. L24: 前進消去を始める．
10. L25〜30: 第 k 行について，式 (7.11) の処理を行う．
11. L25: 式 (7.11) で使う $1/a_{kk}$ を変数 akk に代入する．
12. L27〜30: 第 k 列には 1, 0, ⋯ , 0 を代入すべきであるが，後に影響を与えないので，プログラム上では第 k 列の変更はせず，第 k+1 列以降の処理をする．
13. L31〜37: 式 (7.12) の処理を行う．
14. L38: 前進消去のループの終わり．
15. L39〜44: 後退代入の式 (7.17) の処理．
16. L45: 答えのベクトルを返す．

7.1.2 掃き出し法

ガウスの消去法では前進消去が済むと，係数行列の下三角要素 (a_{ij}, $i > j$) がすべて 0 になっているが，上三角要素 (a_{ij}, $i < j$) に値は残っている．そこで，解を決めるためには後退代入が必要であった．掃き出し法では，各第 k 段階の操作を，式 (7.12) で行 $i = k+1, \cdots, n$ についてではなく $i \neq k$ について

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - a_{ik}^{(k-1)} a_{kj}^{(k)} \qquad (\text{列 } j = k, \cdots, n+1) \tag{7.18}$$

とするので，前進消去が済むと対角要素はすべて 1，非対角要素はすべて 0 となるから，その時点ですべての解が求まっており，後退代入の操作はいらない．図 7.1 (左図) はガウスの消去法の前進消去の過程で必要な要素の変換回数を説明する図である．この立体は底面が n^2 で，a_{nn} から上に伸びる長さ n の垂線の先端を頂点とする正四角錐 (底面積が n^2 で高さが n の四角錐) である．底面の正方形は行列を表す正方形であり，
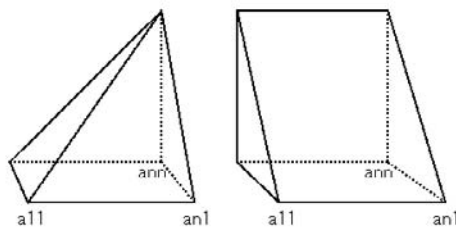
図 7.1　ガウスの消去法 (左図) と掃き出し法 (右図) による要素の変換回数．底面は最初の $n \times n$ の行列であり，下から上に第 k 段階となる．ガウスの消去法による要素の変換回数は，1 辺が n の正方錐となるので，ほぼ $n^3/3$ であるが，掃き出し法によると，横に寝かせた三角柱となるので，ほぼ $n^3/2$ となる．

第 1 段階の操作回数は式 (7.9) からわかるように，ほぼ n^2 である．第 k 段階の操作回数は式 (7.12) からわかるように，ほぼ $(n-k)^2$ であり，図では下から k 段にある正方形に相当する．そこで，ガウスの消去法では要素の変換回数はほぼ $n^3/3$ 回である．一方，掃き出し法 (右図) で必要な要素の変換回数は式 (7.18) からわかるように，$n^3/2$ (底面積が n^2 で高さが n の立方体の半分) であるから，n が大きい場合には，ガウスの消去法のほうが有利である．

7.1.3　LU 分解

式 (7.6) の連立方程式で，係数行列 A が共通で，$\mathbf{b}$ が異なる多数の式を解く場合には，後に述べる A の逆行列 A^{-1} を使って，$\mathbf{x} = A^{-1}\mathbf{b}$ で解を得ることができるが，それは得策ではない．ガウスの消去法では，係数行列 A の下三角要素を消去するときに，$\mathbf{b}$ にも同時に同じ操作を施してきたが，ここでは，消去操作の手順を保存しておき，$\mathbf{b}$ への操作は後で行うことにする．さらに，式 (7.11) や式 (7.12) 等の第 k 段階の操作を大幅に変更する．まず，第 k 行目は変化させず，行 $i = k+1, \cdots, n$ について処理することになる．第 i 行目の第 k 列については

$$a_{ik}^{(k)} = a_{ik}^{(k-1)}/a_{kk}^{(k-1)} \tag{7.19}$$

であり，列 $j = k+1, \cdots, n$ については

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - a_{ik}^{(k)} a_{kj}^{(k-1)} \tag{7.20}$$

とする．この操作によると，行列 A は

$$A^{(n)} = \begin{pmatrix} a_{11}^{(0)} & a_{12}^{(0)} & \cdots & a_{1n}^{(0)} \\ a_{21}^{(1)} & a_{22}^{(1)} & \cdots & a_{2n}^{(1)} \\ & & \cdots & \\ a_{n1}^{(1)} & a_{n2}^{(2)} & \cdots & a_{nn}^{(n-1)} \end{pmatrix} \tag{7.21}$$

となる．この行列 A に出てきた要素を使うと元の A は

$$A^{(0)} = \begin{pmatrix} 1.0 & 0.0 & \cdots & 0.0 \\ a_{21}^{(1)} & 1.0 & \cdots & 0.0 \\ & & \cdots & \\ a_{n1}^{(1)} & a_{n2}^{(2)} & \cdots & 1.0 \end{pmatrix} \begin{pmatrix} a_{11}^{(0)} & a_{12}^{(0)} & \cdots & a_{1n}^{(0)} \\ 0.0 & a_{22}^{(1)} & \cdots & a_{2n}^{(1)} \\ & & \cdots & \\ 0.0 & 0.0 & \cdots & a_{nn}^{(n-1)} \end{pmatrix} = LU \tag{7.22}$$

のように下三角行列 (L) と上三角行列 (U) の積に分解 (LU 分解) することができる．この場合の下三角行列 (L) は，行列 $A^{(n)}$ の下三角部分と全対角要素に 1 を代入したものであり，上三角行列 (U) は，行列 $A^{(n)}$ の上三角部分と全対角要素を代入したものである．そこで，

$$A^{(0)}\mathbf{x} = LU\mathbf{x} = \mathbf{b} \tag{7.23}$$

となるから，$U\mathbf{x} = \mathbf{y}$ と置いて，

$$LU\mathbf{x} = L\mathbf{y} = \mathbf{b} \tag{7.24}$$

とすることができる．まず，$L\mathbf{y} = \mathbf{b}$ から，$\mathbf{y}$ を求めるには，各 $\mathbf{b}$ についても，A について行った消去法の操作

$$\begin{aligned} y_1 &= b_1 \\ y_i &= b_i - \sum_{j=1}^{i-1} a_{ij}^{(j)} y_j \qquad (i = 2, \cdots, n) \end{aligned} \tag{7.25}$$

を行うことになる．続けて $U\mathbf{x} = \mathbf{y}$ から $\mathbf{x}$ を求めるには，式 (7.17) の後退代入の操作を行えばよい．

プログラム (LU_Bunkai.java) のメソッド (lu_Bunkai) と，その説明は以下のとおりである．ピボット選択を列についても行っている．

LU_Bunkai.java のメソッド (lu_Bunkai)：

```
 1    public double[][] lu_Bunkai(int n, double[][] a,
 2      int m, double[][]  b) {
 3    double[][] x= new double[n][m];
 4    int[] ipiv= new int[n];
 5    double[] winv= new double[n];
 6    int ip, ipw, iw;
 7    double w;
 8    for (int k=0; k<n; k++) {
 9      ipiv[k]= k;
10      w= 0.0;
11      for (int i=0; i<n; i++) {
12        if (Math.abs(a[k][i])  >  w) {
13          w = Math.abs(a[k][i]);
14        }
15      winv[k]= 1/w;
16      }
17    }
```

```
18     for (int k=0; k<n; k++) {
19       w= 0.0;
20       ip= 0;
21       for (int i=k; i<n; i++) {
22         ipw= ipiv[i];
23         if (Math.abs(a[ipw][k])*winv[ipw]  >  w) {
24           w = Math.abs(a[ipw][k])*winv[ipw];
25           ip = i;
26         }
27       }
28       ipw= ipiv[ip];
29       if (ip != k) {
30         ipiv[ip]= ipiv[k];
31         ipiv[k]= ipw;
32       }
33       double aipwk= 1.0/a[ipw][k];
34       double aipk;
35       for (int i=k+1; i<n; i++) {
36         ip= ipiv[i];
37         a[ip][k] *= aipwk;
38         aipk= a[ip][k];
39         for (int j=k+1; j<n; j++) {
40           a[ip][j] -= aipk*a[ipw][j];
41         }
42       }
43     }
44     for (int k=0; k<m; k++) {
45       for (int i=0; i<n; i++) {
46         ip= ipiv[i];
47         w= b[ip][k];
48         for ( int j=0; j<i; j++) {
49           w -= a[ip][j]*x[j][k];
50         }
51         x[i][k]= w;
52       }
53       for (int i=n-1; i>=0; i--) {
54         w= x[i][k];
55         ip= ipiv[i];
56         for ( int j=i+1; j<n; j++) {
57           w -= a[ip][j]*x[j][k];
58         }
59         x[i][k]= w/a[ip][i];
60       }
61     }
62     return x;
63   }
```

1. L1, 2〜63: メソッド lu_Bunkai の定義である．多重連立方程式は $a_{ij} * x_{jk} = b_{ik}$ で，戻り値は解の行列 x である．引数の n は係数行列 a の次元，a は係数行列，m は右辺の定数行列 b の列の数であり，係数行列を a とする連立方程式の数で

ある．b は右辺の定数行列である．

2. L3: x は解の行列である．
3. L4: ピボット入れ替えを記録しておくために列番号を代入する配列である．
4. L5: ピボットの要素の逆数を代入しておく配列である．
5. L8〜17: 各行ごとに絶対値が最大の要素を探し，その逆数を配列 winv に代入する．
6. L18〜43: 一番外側のループであり，第 k 段階に関するループをまわす．
7. L19〜32: ピボット選択を行う．
8. L33〜42: 式 (7.19) と式 (7.20) の処理を行う．
9. L43: L18 からのループの終わりで，LU 分解が済んでいる．ピボット選択を使っているから，順番が入れ替わっているが，そのことも考慮すれば，配列 A の下三角の部分には L があり，対角要素と上三角部分には U がある．
10. L44〜61: b の列に関するループである．
11. L45〜52: 式 (7.25) の処理を行う．
12. L53〜60: 式 (7.17) の後退代入の処理を行う．
13. L61: b の列に関するループの終了．
14. L62: 答えのベクトルを返す．

7.2 行列式

連立 1 次方程式の解法には，以上述べた方法のほかに，クラマーの公式がある．これは数式としては簡潔で美しいが，計算プログラムでのステップ数が $(n+1)n!$ であるから消去法の $n^3/3$ と比較すると，次元数 n が大きい場合には圧倒的に効率が悪い．しかし，次元数が 2, 3 の場合には役に立ち，筆算でも使われる．実際，式 (3.3) の導出にも使われている．

7.2.1 クラマーの公式

n 元連立 1 次方程式は式 (7.6) である．行列 A の行列式

$$\det|A| = \begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ & & \cdots & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix} \tag{7.26}$$

を使うと，連立方程式の解は，

$$x_i = \frac{\det|A_i|}{\det|A|} \tag{7.27}$$

である．分子の $\det|A_i|$ は係数行列 A の第 i 列と，連立方程式の右辺のベクトル $\mathbf{b}$ とを入れ替えた行列の行列式

$$\det|A_i| = \begin{vmatrix} a_{11} & a_{12} & \cdot & b_1 & \cdot & a_{1n} \\ a_{21} & a_{22} & \cdot & b_2 & \cdot & a_{2n} \\ & & \cdot & & \cdot & \\ a_{n1} & a_{n2} & \cdot & b_n & \cdot & a_{nn} \end{vmatrix} \tag{7.28}$$

である．

ここで気が付くことは，式 (7.27) の分母である式 (7.26) の値がゼロになると，解を求められないことである．このように係数行列の行列式がゼロになると，消去法や掃き出し法においては，計算の途中でピボット選択をしてもピボットがゼロになってしまい，計算ができなくなってしまう．一般的に，行列式の値がゼロでない行列を正則行列という．また，正則行列には逆行列が存在する．

7.2.2　行列式の値

(2×2) 型の行列の行列式は

$$\det A = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{21}a_{12} \tag{7.29}$$

であり，(3×3) 型の行列の行列式は

$$\det A = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}$$
$$= a_{11}a_{22}a_{33} + a_{31}a_{12}a_{23} + a_{21}a_{32}a_{13} - a_{11}a_{32}a_{23} - a_{31}a_{22}a_{13} - a_{21}a_{12}a_{33} \tag{7.30}$$

である．そして，$(n \times n)$ 型の行列の行列式の一般的な定義は

$$\det A = \sum_{(j_1,\, j_2,\, \cdots,\, j_n)} \sigma(j_1,\, j_2,\, \cdots,\, j_n)\, a_{j_1 1} a_{j_2 2} \cdots a_{j_n n} \tag{7.31}$$

である．ここで使われた関数 $\sigma(j_1,\, j_2,\, \cdots,\, j_n)$ は，$(1,\, 2,\, \cdots,\, n)$ の順列の一つである $(j_1,\, j_2,\, \cdots,\, j_n)$ について，$(j_1,\, j_2,\, \cdots,\, j_n)$ が $(1,\, 2,\, \cdots,\, n)$ から偶数回の互換によって作られるなら $\sigma(j_1,\, j_2,\, \cdots,\, j_n) = +1$ であり，奇数回の互換によって作られるなら $\sigma(j_1,\, j_2,\, \cdots,\, j_n) = -1$ である．また，$\sum$ で表される和は，すべての順列 $(j_1,\, j_2,\, \cdots,\, j_n)$ について加えることを意味している．

(2×2) 型の行列は 2 本の列 (縦) ベクトルあるいは行 (横) ベクトルからできていると見ることが可能であるが，紙面の都合上，行ベクトルのほうを使うこととする．行列式は 2 本のベクトルの張る平行四辺形の面積

$$\det A = |\mathbf{a}_1||\mathbf{a}_2| \sin\theta \tag{7.32}$$

に等しい．ここで θ は，ベクトル $\mathbf{a}_1 = (a_{11},\, a_{12})$ からベクトル $\mathbf{a}_2 = (a_{21},\, a_{22})$ までの角度であり，x, y 面上で反時計回りを正とする．

図 7.2 (左図) は，行ベクトルを $\mathbf{a}_1 = (a_{11}, a_{12})$，$\mathbf{a}_2 = (a_{21}, a_{22})$ とした行列 $\begin{pmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \end{pmatrix}$ の行列式の図である．平行四辺形の面積が式 (7.29)，すなわち，破線の矩形から点線の矩形を引いたものに等しいことは，この図にもう少し補助線を加えて平面幾何の問題として考えれば理解できる．

(3×3) 型の行列の場合は 3 本のベクトルの張る平行六面体の体積に等しい．ただし，ベクトルの張る順番によっては負の値となる．図 7.2 (右図) は行ベクトルを $\mathbf{a}_1 = (a_{11}, a_{12}, a_{13})$，$\mathbf{a}_2 = (a_{21}, a_{22}, a_{23})$，$\mathbf{a}_3 = (a_{31}, a_{32}, a_{33})$ とした行列 $\begin{pmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \mathbf{a}_3 \end{pmatrix}$ の行列式の図である．平行六面体の体積が式 (7.30) に等しいことは，(2×2) 型の場合ほど簡単ではない．原点で座標軸の回転を行っても 3 本のベクトルの相互の関係は不変なので，体積は変わらないから座標軸の回転を使う．2 本のベクトル $\mathbf{a}_1 = (a_{11}, a_{12}, a_{13})$，$\mathbf{a}_2 = (a_{21}, a_{22}, a_{23})$ を含む面上に新しい x', y' 軸をとり，その面に垂直に z' 軸をとる．すると，平行六面体の体積は底面の平行四辺形 $(a'_{11}a'_{22} - a'_{21}a'_{12})$ にベクトル $\mathbf{a}_3$ の z' 成分 a'_{33} を掛けたものになるから，式 (7.30) の $a'_{11}a'_{22}a'_{33} - a'_{21}a'_{12}a'_{33}$ に等しくなる．そして，残りの 4 項はゼロとなっているから，式 (7.30) は平行六面体の体積である．

一般的に，$(n \times n)$ 行列の行列式の値は，n 次元空間で n 本のベクトル $\mathbf{a}_1$，$\mathbf{a}_2$，$\cdots$，$\mathbf{a}_n$ の張る超平行面立体の体積となる．そこで，もし n 本のベクトルが n 次元空間を張らず，$n-1$ 次元以下の空間に収まってしまうと，n 次元空間での体積はゼロになってしまい，行列式の値もゼロである．ちなみに，n 本のベクトルの張る空間が m 次元の場合には，その行列のランク (位) は m であるという．また，$(n \times n)$ 行列のランクが n の場合には，どのベクトルも他のベクトルの 1 次結合では表せないから，ベクトルは互いに 1 次独立であるといい，n 未満の場合には 1 次従属であるという．

行列式を求めるプログラムに関係して必要な行列式の性質を次に述べる．行という文字を使うが，すべてを列という文字に置き換えても成り立つ．直感的に理解するために (3×3) 型の図で説明する．これらの証明は，もちろん式 (7.31) を使って，一般的に行うことができる．
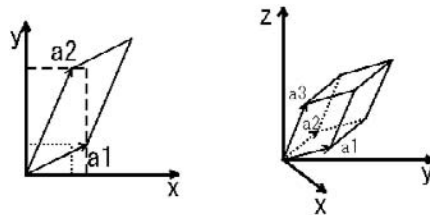
図 7.2　2 行 2 列の行列 (左図) と 3 行 3 列の行列 (右図) の行列式の値となる平行四辺形と平行六面体．

(1)　ある二つの行同士の入れ替えをすると，体積(行列式の値)の符号が変わる．
(2)　j 行が i 行に等しかったら(1)により，あるいは厚みがないので，体積(行列式の値)はゼロである．
(3)　ある行に数 c を掛けるとベクトルの長さが c 倍になるから，体積(行列式の値)は c 倍になる．
(4)　ある行に他の行の d 倍を加えても，体積(行列式の値)は変わらない．

そこで，行列式の値を求める行列に，連立方程式の係数行列と同じ消去法を適用することができるが，次のことに注意しておく必要がある．ピボットの入れ替えがあったら，(1)により行列式の値の符号を反転する．式(7.11)によると，各 k 段階でピボット行が $a_{kk}^{(k-1)}$ で割られて対角要素は1になっているから，(3)により，最後に $a_{kk}^{(k-1)}$ を掛けなければならない．また，他の i 行にはピボット行の $a_{ik}^{(k-1)}$ 倍したものが足されているが，(4)により，その影響はゼロである．最後の行列の行列式の値は1となるから，元の行列式の値は各段階で割り算に使ったピボット $a_{kk}^{(k-1)}$ のすべてを1に掛ければよく，それは

$$\det |A| = 1 \prod_{k=1}^{n} a_{kk}^{(k-1)} \tag{7.33}$$

である．

プログラム(LU_Gyouretsushiki.java)のメソッド(lu_Gyouretsushiki)はプログラム(LU_Bunkai.java)のメソッド(lu_Bunkai)とLU分解の部分は共通であり，式(7.33)の操作が追加されているだけである．その部分と説明は以下のとおりである．

LU_Gyouretsushiki.javaのメソッド(lu_Gyouretsushiki)の一部：

```
 1    public double lu_Gyouretsushiki(int n, double[][] a) {
 2      double det= 1.0;
 3      int[] ipiv= new int[n];
 4    中略
 5      if (ip != k) {
 6        ipiv[ip]= ipiv[k];
 7        ipiv[k]= ipw;
 8        det= -det;
 9      }
10      double aipwk= 1.0/a[ipw][k];
11      det /= aipwk;
12      double aipk;
13    中略
14      return det;
15    }
```

1. L1〜15: メソッド lu_Gyouretsushiki の定義である．プログラム(LU_Bunkai.java)のメソッド(lu_Bunkai)とほとんど同じである．メソッド(lu_Bunkai)のL1〜3と入れ替わっている．引数の n は行列 a の次元，a は行列式の値を求める配列である．

2. L2: 行列式の値の計算のための変数 det の初期化.
3. L5〜9: ピボットの入れ替えがあったら，符号を反転する.
4. L10〜12: 対角要素を掛ける.
5. L14: det の値を返している.

7.3 逆行列

掃き出し法で前進消去が済むと，元の係数行列 A は単位行列になる．式 (7.16) の O は消去法の変換操作であるから，掃き出し法の変換操作とは少し異なるが，掃き出し法の変換操作 O を使えば，

$$A^{(n)} = OA = I \tag{7.34}$$

となる．この変換操作 O の行列を作れば，それがまさに逆行列であるが，この方法でそのまま計算することは効率が良くないので，掃き出し法があるわけであり，それを使うことになる．そこで，O を掃き出し法の変換操作として単位行列に作用させるために，同じ過程を単位行列に施せば，

$$OI = O \tag{7.35}$$

となる．すると，行列 O，すなわち A の逆行列が得られる．式 (7.6) でベクトル $\mathbf{x}$ と右辺のベクトル $\mathbf{b}$ を $(n \times n)$ の行列に拡張して行列 b を単位行列にすることになる．

そこで，行列 A の $(n \times 2n)$ 行列のほかに，$(n \times n)$ の単位行列を b として，メソッド lu_Bunkai を呼べば戻り値が逆行列となっている．

プログラム (LU_GyakugyouretsuTest.java) のメソッド (main) の一部と，その説明は以下のとおりである．

LU_GyakugyouretsuTest.java のメソッド (main) の一部：

```
1   public static void main(String args[]){
2     double[][] a= {{1, 4, 3}, {2, 5, 4}, {1, -3, -2}};
3     double[][] i= {{1,0,0}, {0,1,0}, {0,0,1}};
4     int n= a.length;
5     double[][] x= new double[n][n];
6   中略
7     LU_Bunkai gs= new LU_Bunkai();
8     x= gs.lu_Bunkai(n, a, n, i);
9   中略
10  }
```

1. L1〜10: メソッド main の定義である.
2. L2〜5: a は逆行列を求める元の行列，i は単位行列，x は逆行列を代入する行列である.
3. L7: LU_Bunkai 型のインスタンスを作る.
4. L8: 多重連立方程式を解いて，逆行列を求める.

演習問題

7.1　次の連立方程式を解きなさい.

$$\left.\begin{aligned} 2x+y+z&=-1\\ x+2y+z&=-2\\ x+y+2z&=-2 \end{aligned}\right\}$$

答： $x=0.25,\quad y=-0.75,\quad z=-0.75$

7.2　次の行列式の値を求めなさい.

$$\begin{vmatrix} 1 & 3 & 2 & -1\\ 2 & 0 & 1 & -2\\ -1 & 5 & 1 & 1\\ 2 & 7 & -6 & 3 \end{vmatrix}$$

答： 30

第 8 章

関数近似と平滑化

実験値には多かれ少なかれ誤差が含まれるから，実験を繰り返すたびに少しずつ異なった値が得られる．そこで，最も確からしい値 (最確値) を決定することが必要になる．変数 x に対して，データ量 y が得られるとき，x と y の関係 $y(x)$ が，何かある関数の形になることがわかっているときには，その関数のパラメータを決めればよいことになる．そのパラメータを決める方法に最小二乗法という方法がある．一方，関数型がわからない場合には，すべてのデータ点 x について，その近傍の数点の値の平均をとるなどして，データのばらつきを滑らかにする平滑化 (smoothing) をすることになる．

数値計算において，その関数型が解析的にわかっているけれども，複雑すぎて変数 x が変わるたびに毎回計算するわけにいかないような場合には，チェビシェフ近似が役に立つ．それは，多項式近似ができる場合であり，n 次の多項式 $P_n(x)$ で近似するなら，$n+1$ 次のチェビシェフ多項式のゼロ点をデータ点 (標本点) として，元となる複雑な関数の値を計算しておけばよい．一方，存在するデータ点のすべてを使って多項式などを作るのではなく，計算する点 x の周りの数個のデータ点だけを局所的に使う方法として，スプライン関数や，2 次元曲面の場合のベジェ曲面などがある．第 3 章で述べたラグランジュの補間法なども，関数近似の中に入れることができるであろう．

8.1 最小二乗法

ある量 x の値を n 回繰り返し求めたら，$x_1, x_2, \cdots, x_n$ であった．真の値はわからないので，最確値 x_0 を決定しなければならない．各測定値と最確値の差 $\Delta x_i = x_i - x_0$ の発生がまったく偶然であるとすると，その分布は通常，正規分布

$$(2\pi)^{-\frac{1}{2}} e^{-\frac{\Delta x^2}{2}} \tag{8.1}$$

となるので，n 回での差が $\Delta x_1, \Delta x_2, \cdots, \Delta x_n$ となる確率は正規分布関数の，

$$f(\Delta x_1, \Delta x_2, \cdots, \Delta x_n) = (2\pi)^{-\frac{n}{2}} e^{-(\Delta x_1^2 + \Delta x_2^2 + \cdots + \Delta x_n^2)/2} \tag{8.2}$$

である．最確値は，この確率を最も大きくする値であるから，式 (8.2) の指数部 $S = \Delta x_1^2 + \Delta x_2^2 + \cdots + \Delta x_n^2$ を最小にすることから求められる．すなわち，

$$S = \Delta x_1^2 + \Delta x_2^2 + \cdots + \Delta x_n^2 = \sum_{i=1}^{n}(x_i - x_0)^2 \tag{8.3}$$

の未知数 x_0 での微分がゼロ，

$$\frac{dS}{dx_0} = -2\sum_{i=1}^{n}(x_i - x_0) = 0 \tag{8.4}$$

である．これから，

$$x_0 = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{8.5}$$

となる．S は最確値と各データ値との差の 2 乗の和であるから，この最確値の求め方は最小二乗法といわれ，この場合には最確値は算術平均値である．

8.1.1　1 個の変数に対して線形関係がある場合

ある量 x の値に対し測定値 y が x と線形関係にあることがわかっている場合には，最適関数

$$y(x) = ax + b \tag{8.6}$$

の係数 a と b を決めることになる．n 回の測定データの組を $\{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$ とする．この場合には，x の値には曖昧さがなく y の値にのみばらつきがあると考えるので，2 乗和は

$$S = \sum_{i=1}^{n}\{y_i - (ax_i + b)\}^2 \tag{8.7}$$

となる．最小二乗法では未知数 a と b による式 (8.7) の偏微分が 0 となることから求めるので，そこで，

$$\frac{\partial S}{\partial a} = 0, \quad \frac{\partial S}{\partial b} = 0 \tag{8.8}$$

より，

$$2\sum_{i=1}^{n}\{x_i^2 a + x_i b - x_i y_i\} = 0, \quad 2\sum_{i=1}^{n}\{x_i a + b - y_i\} = 0 \tag{8.9}$$

となる．これは整理すると，未知数 a, b に対する連立方程式

$$[x^2]a + [x]b = [xy], \quad [x]a + [1]b = [y] \tag{8.10}$$

である．ここで，

$$\begin{aligned} &[x^2] \equiv \sum_{i=1}^{n} x_i^2, \quad [x] \equiv \sum_{i=1}^{n} x_i, \quad [xy] \equiv \sum_{i=1}^{n} x_i y_i \\ &[1] \equiv \sum_{i=1}^{n} 1 = n, \quad [y] \equiv \sum_{i=1}^{n} y_i \end{aligned} \tag{8.11}$$

を使った．式 (8.10) は未知数 a と b に関する連立方程式であるから，それを解くと，

$$a = \frac{n[xy] - [x][y]}{n[x^2] - [x]^2}, \quad b = \frac{[x^2][y] - [x][xy]}{n[x^2] - [x]^2} \tag{8.12}$$

が決まる．

図 8.1 は $[0, 9.5]$ で等間隔な 20 点の x の値に対する y の値を $y = 0.9x+1.0+\epsilon$ で作った 20 組のデータについて，最小二乗法で求めた 1 次式のグラフである．ϵ は $\sigma = 0.9$ の正規分布乱数である．プログラム (Saishou_Jijou_Hou.java) のメソッド (saishou_Jijou_Hou) と，その説明は以下のとおりである．

Saishou_Jijou_Hou.java のメソッド (saishou_Jijou_Hou)：

```
 1  public double[] saishou_Jijou_Hou(int m, int n, double[][] x,
          double[] y){
 2    double[][] a = new double[m][m];
 3    double[] b = new double[m];
 4    double[] p = new double[m];
 5    double s;
 6    for (int i=0; i<m; i++) {
 7      for (int j=0; j<=i; j++) {
 8        s= 0.0;
 9        for (int k=0; k<n; k++) {
10          s += x[k][i]*x[k][j];
11        }
12        a[i][j]= a[j][i]= s;
13      }
14    }
15    for (int i=0; i<m; i++) {
16      s= 0.0;
17      for (int k=0; k<n; k++) {
18        s += x[k][i]*y[k];
19        }
20      b[i]= s;
```
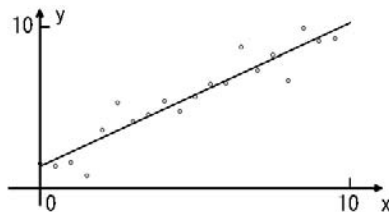
図 8.1　線形関係のあるデータの最小二乗法による 1 次式の決定．○ 印は，$[0, 9.5]$ で等間隔な 20 点の x の値に対する y の値を $y = 0.9x + 1.0 + \epsilon$ で作った 20 組のデータであり，直線は最小二乗法で決定した 1 次式である．ϵ は $\sigma = 0.9$ の正規分布乱数である．この場合は，$a = 0.89,\ b = 1.32$ である．（Z08_01_Saishou_Jijou_Hou.java）

```
21      }
22      Gauss_Shoukyohou gs= new Gauss_Shoukyohou();
23      p= gs.gauss_Shoukyohou(m,a,b);
24    中略
25      return p;
26    }
```

1. L1〜26: メソッド saishou_Jijou_Hou の定義である．引数の m は自由度で近似方程式の係数の数，n はデータの組数である．x はデータの配列であり，第 1 添字はデータの組数 n，第 2 添字は自由度 m である．y は関数値の配列で添字はデータの組数 n である．戻り値は近似方程式の係数の配列である．
2. L2: 近似方程式の係数を求めるための，連立方程式の係数行列である．
3. L3: 連立方程式の右辺の定数配列である．
4. L4: 近似方程式の係数を代入する配列である．
5. L6〜14: 式 (8.11) により，連立方程式の係数行列を作成する．
6. L15〜21: 式 (8.11) により，連立方程式の右辺の定数を作成する．
7. L22: Gauss_Shoukyohou クラスのインスタンスを作る．
8. L23: 連立方程式を解く．
9. L25: 近似方程式の係数を戻り値として返す．

8.1.2　変数が 2 個以上ある場合

変数として x, y の 2 個がある場合には，最適関数

$$z(x, y) = ax + by + c \tag{8.13}$$

の係数 a, b, c を決めることになる．1 回の測定で求められる量は x, y, z の 3 量 (自由度が 3) であるから，n 回の測定データの組は $\{(x_1, y_1, z_1), (x_2, y_2, z_2), \cdots, (x_n, y_n, z_n)\}$ となる．係数 a, b, c は最小二乗法によると，

$$S = \sum_{i=1}^{n} \{z_i - (ax_i + by_i + c)\}^2 \tag{8.14}$$

の a, b, c による偏微分がそれぞれ 0 となることから得られる連立方程式,

$$\begin{aligned} &[x^2]a + [xy]b + [x]c = [xz] \\ &[xy]a + [y^2]b + [y]c = [yz] \\ &[x]a + [y]b + [1]c = [z] \end{aligned} \tag{8.15}$$

を解いて求められる．

より一般的に，変数が m 個で自由度が $m+1$ の場合には，最適関数

$$y(x^{(1)}, \cdots, x^{(m)}) = \sum_{i=1}^{m} a^{(i)} x^{(i)} + a^{(m+1)} \tag{8.16}$$

の $a^{(i)}$ $(i=1,\cdots,m+1)$ を決めなければならない．この場合の連立方程式は，

$$\begin{aligned}&\sum_{i=1}^{m}[x^{(i)}x^{(j)}]a^{(i)}+[x^{(j)}]a^{(m+1)}=[x^{(j)}y] \qquad (j=1,\cdots,m)\\&\sum_{i=1}^{m}[x^{(i)}]a^{(i)}+[1]a^{(m+1)}=[y]\end{aligned} \tag{8.17}$$

となる．

8.1.3　変数 x の 2 次式の場合

x の 2 次式の場合には，最適関数は $y(x)=ax^2+bx+c$ であるから自由度は 3 であり，式 (8.15) において，$x\to x^2$，$y\to x$，$z\to y$ の置き換えをすればよく，連立方程式は

$$\begin{aligned}&[x^4]a+[x^3]b+[x^2]c=[x^2y]\\&[x^3]a+[x^2]b+[x]c=[xy]\\&[x^2]a+[x]b+[1]c=[y]\end{aligned} \tag{8.18}$$

である．

図 8.2 は $[-1,1]$ で等間隔な 41 点の x の値に対する y の値を $y=x^2+x+1.0+\epsilon$ で作った 41 組のデータについて，最小二乗法で求めた 2 次式のグラフである．ϵ は $\sigma=0.1$ の正規分布乱数である．
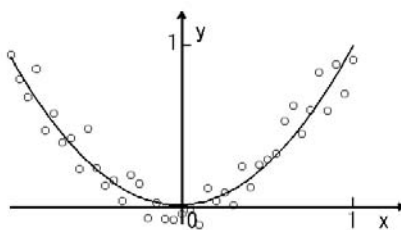
図 8.2　2 次式の関係があるデータの最小二乗法による 2 次式の決定．○印は，$[-1,1]$ で等間隔な 41 点の x の値に対する y の値を $y=x^2+x+1.0+\epsilon$ で作った 41 組のデータであり，実線は最小二乗法で決定した 2 次式である．ϵ は $\sigma=0.1$ の正規分布乱数である．この場合は，$y=0.948x^2+0.035x+0.012$ である．（Z08_02_SJH_2jishiki.java）

8.2　チェビシェフ多項式近似

チェビシェフ多項式 $T_n(x)$ の重要な特性は次のとおりである．いま，区間 $-1 \leq x \leq 1$ で関数 $f(x) \equiv 0$（恒等的にゼロ）を最高次の係数が 1 である n 次多項式で近似する場合には，その最良近似は $\dfrac{1}{2^{n-1}}T_n(x)$ で与えられる．

チェビシェフ多項式の定義は

$$\begin{aligned} T_n(x) &= \cos(n\cos^{-1}x) \\ &= \cos(n\theta) \qquad \text{ただし } \cos\theta = x \end{aligned} \tag{8.19}$$

である．そこで，$T_n(x) = 0$ を満たす条件は $n\theta = \left(k - \frac{1}{2}\right)\pi$ $(k = 1, 2, \cdots, n)$ であるから，$x = x_k = \cos\{(2k-1)\pi/2n\}$ のときである．この n 個の点では $T_n(x) = 0$ であるから，$f(x) \equiv 0$ の関数と一致している．関数 $f(x) \equiv 0$ の近似として最も大きな誤差は $T_n(x) = \pm 1$ であり，それは $n\theta = k\pi$ $(k = 0, 1, \cdots, n)$ のときであるから，$x = x_k = \dfrac{\cos k\pi}{n}$ のときとなる．そこで，多項式近似ができるような一般的な関数 $g(x)$ を n 次の多項式 $P_n(x)$ で近似するときには，標本点として，$T_{n+1}(x) = 0$ となる $(n+1)$ 個の点 x_k を用いることにする．n 次の多項式

$$P_n(x) = \sum_{i=0}^{n} c_i x^i \tag{8.20}$$

の $n+1$ 個の係数 c_i は，その $n+1$ 個の点で $g(x_k)$ をできるだけ正確に求め，$n+1$ 個の標本点で成り立つ関係式の連立 1 次方程式を解くことによって決定することができる．この近似多項式の誤差の目安とするためには，$T_{n+1}(x) = 1$ となる $n+2$ 個の x_k 点において，$g(x_k)$ と $P_n(x_k)$ の差を求めればよい．

チェビシェフ近似の要点は，次のとおりである．n 次の近似多項式を作るときには，最低 $n+1$ 個の異なった標本点が必要であるが，その点のとり方はまったく任意である．ただ一般的にいうと，点を細かくとった部分は近似が良く，粗くとった部分は悪くなる．そこで，点を等間隔にとってもよいであろうが，外側に標本点がない両端の部分で近似が悪くなってしまうので，チェビシェフ近似では両端に近づくほど点の間隔が細かくなり，精度が均一になるようになっている．

図 8.3 は 1 周期の三角関数の，3 〜 6 次式までのチェビシェフ多項式での近似を示すものである．3 次式から 6 次式に進むに従って近似が良くなり，図に使う程度であったら，6 次式で十分精度があるといえよう．表 8.1 は図 8.3 にマークで示した近似値の誤差を数値で示すものである．x は座標値であり，Δy は誤差である．3 〜 6 次式とも，偶数番目の点は標本点であるから誤差は小さく計算機の精度程度である．奇数番目の点は近似誤差が極大になる点であり，3 次式ではかなり大きいが，4 次式，5 次式，6 次式へと進むにつれて精度が良くなっている．実際に使うときは，必要とする精度が出るまで次数を上げて試す必要がある．また，x の範囲が広いときには区間を分けて，近似式を別々に作るほうが計算時間を少なくすることができる．

図 8.3　三角関数のチェビシェフ多項式による近似．曲線は正確なグラフであり，マークが近似値である．実線と + 印は 3 次式，点線と ∗ 印は 4 次式，破線と ∘ 印は 5 次式，一点鎖線と × 印は 6 次式による近似の様子である．n 次式の場合には，印は $(2n+3)$ 個ある．標本点は $n+1$ 点あり，それぞれ左から偶数番目の点が該当し，誤差はない．両端を含めた奇数番目の点は，近似誤差が極大になる点であり $n+2$ 点ある．(Z08_03_tchebycheff.java)

表 8.1　三角関数のチェビシェフ多項式による近似精度．それぞれ偶数番目の点は標本点であるから丸め誤差だけであり，奇数番目の点は近似誤差が極大になる点である．(Z08_03_tchebycheff.java)

3 次式		4 次式		5 次式		6 次式	
x	Δy	x	Δy	x	Δy	x	Δy
0.0	0.1953	0.0	0.0629	0.0	−0.0279	0.0	6.88E−5
0.2391	3.88E−16	0.1537	−3.05E−16	0.1070	−4.44E−16	0.0787	4.44E−16
0.9201	−0.1567	0.5999	−0.0737	0.4208	0.0279	0.3111	1.97E−4
1.9393	1.11E−16	1.2950	1.11E−16	0.9201	−2.49E−16	0.6853	5.55E−16
3.1415	−9.95E−17	2.1707	0.0989	1.5707	−0.0267	1.1828	−9.85E−4
4.3438	−3.33E−16	3.1415	5.55E−17	2.3284	−2.22E−16	1.7785	3.33E−16
5.3630	0.1567	4.1123	−0.1144	3.1415	0.0225	2.4425	0.0021
6.0440	−5.82E−16	4.9881	2.22E−16	3.9546	1.83E−16	3.1415	0.0
6.2831	−0.1953	5.6831	0.1103	4.7123	−0.0163	3.8406	−0.0034
		6.1294	4.44E−16	5.3630	−4.44E−16	4.5046	1.66E−16
		6.2831	−0.1053	5.8622	0.0111	5.1003	0.0043
				6.1761	−7.77E−16	5.5977	2.22E−16
				6.2831	−0.0093	5.9720	−0.0047
						6.2044	6.66E−16
						6.2831	0.0049

プログラム (Tchebycheff.java) のコンストラクタ，およびメソッドの構成は以下のとおりである．

Tchebycheff.java のコンストラクタおよびメソッドの構成：

```
 1 public class Tchebycheff {
 2   int norder, nfunc;
 3   double cm, cp;
 4   double[][] c;
 5
 6   public Tchebycheff(double xmin, double xmax, int nfunc,
 7       int norder, boolean check, MyFunction2 fu) {
 8   中略
 9   }
10
11   public double[] checkPrint(MyFunction2 fu, boolean print) {
12   中略
13   }
14
15   public double[][] tchebycheff(double[] x) {
16   中略
17   }
18 }
```

1. L1〜18: クラス Tchebycheff の定義である．
2. L2〜4: クラス内のコンストラクタとメソッドで共通に使うフィールドの宣言である．norder は近似式の次数，nfunc は近似する関数の数である．cm, cp は x 座標の座標変換の定数であり，c は近似関数の係数である．第 1 添字が関数の数 nfunc，第 2 添字は係数の数 norder+1 である．
3. L6, 7: コンストラクタ Tchebycheff の定義である．引数の xmin は近似する領域の最小値，xmax は最大値である．nfunc は近似する関数の数，norder は近似式の次数，check は誤差の数値の出力の制御で true なら出力をする．fu はインタフェース MyFunction2 のインスタンスであり，近似する関数のメソッドが定義されている．
4. L11, 12: メソッド checkPrint の定義である．引数の fu はインタフェース MyFunction2 のインスタンスである．print は精度を示す数値の出力の制御で true なら出力をする．
5. L15: メソッド tchebycheff の定義である．引数の x は近似値を求める x 座標の配列である．戻り値は近似値の配列であり，第 1 引数が関数の数，第 2 引数が係数の数の 2 次元配列である．

プログラム (Tchebycheff.java) のコンストラクタ (Tchebycheff) と，その説明は以下のとおりである．

Tchebycheff.java のコンストラクタ (Tchebycheff)：

```
1   public Tchebycheff(double xmin, double xmax, int nfunc,
2       int norder, boolean check, MyFunction2 fu) {
3     this.nfunc= nfunc;
```

```
 4     this.norder= norder;
 5     cm= (xmax-xmin)/2.0;
 6     cp= cm+xmin;
 7     c= new double[nfunc][norder+1];
 8     double[] x= new double[norder+1];
 9     double[] xt= new double[2*norder+3];
10     double[][] f;
11     double[][] a= new double[norder+1][norder+1];
12     double[] b= new double[norder+1];
13     double w, cw;
14     int n= 0;
15     double a1= Math.PI*0.5/(norder+1);
16     for (int k=0; k<=norder; k++) {
17       x[k]= cm*Math.cos((2*(norder-k)+1)*a1)+cp;
18     }
19     f= fu.function(x,nfunc,norder+1);
20     Gauss_Shoukyohou gs= new Gauss_Shoukyohou();
21     for (int l=0; l<nfunc; l++) {
22       for (int i=0; i<=norder; i++) {
23         b[i]= f[l][i];
24         w= 1.0;
25         cw= x[i]-cp;
26         for (int k=0; k<=norder; k++) {
27           a[i][k]= w;
28           w= w*cw;
29         }
30       }
31       c[l]= gs.gauss_Shoukyohou(norder+1,a,b);
32     }
33   中略
34   }
```

1. L1, 2〜34: コンストラクタ Tchebycheff の定義である．引数の説明は先のとおりである．
2. L5, 6: cm，cp に x 座標の幅の半分と中央の座標値を代入する．
3. L7: 近似関数の係数を代入する配列 c のインスタンスを作る．
4. L8: チェビシェフ多項式の値がゼロとなる点の x 座標の配列である．
5. L9: 精度を調べるための，チェビシェフ多項式の値がゼロとなる点と，その合間で近似が悪くなる点の x 座標の配列である．
6. L10: 関数値の配列である．
7. L11: 近似式の係数を決める連立 1 次方程式の係数行列である．
8. L12: 近似式の係数を決める連立 1 次方程式の右辺の定数の配列である．
9. L16〜18: チェビシェフ多項式の値がゼロとなる点の x 座標を求める．
10. L19: その x 座標について，関数値を求める．
11. L20: ガウスの消去法のクラス Gauss_Shoukyohou のインスタンスを作る．
12. L21〜32: 近似関数の数だけループをまわす．
13. L22〜30: 連立 1 次方程式の右辺の定数と係数行列を式 (8.20) で求める．
14. L31: メソッド gauss_Shoukyohou で近似式の係数を決める．

プログラム (Tchebycheff.java) のメソッド (tchebycheff) と，その説明は以下のとおりである．

Tchebycheff.java のメソッド (tchebycheff)：

```
 1  public double[][] tchebycheff(double[] x) {
 2    int npoint= x.length;
 3    double[][] f= new double[nfunc][npoint];
 4    double xk;
 5    double w;
 6    for (int k=0; k<=npoint-1; k++) {
 7      xk= x[k];
 8      for (int j=0; j<nfunc; j++) {
 9        w= c[j][norder];
10        for (int l=norder-1; l>=0; l--) {
11          w= (xk-cp)*w+c[j][l];
12        }
13        f[j][k]= w;
14      }
15    }
16    return f;
17  }
```

1. L1〜17: メソッド tchebycheff の宣言である．引数の説明は先のとおりである．
2. L3: 戻り値となる近似関数値の配列の定義である．
3. L6〜15: 近似値を求める座標の数だけループをまわす．
4. L8〜14: 求める近似関数の数だけループをまわす．
5. L10〜12: 式 (8.20) で近似値を計算する．
6. L16: 近似値の配列を返す．

インタフェース (MyFunction2.java) の抽象メソッド (function) は以下のとおりである．

MyFunction2.java の抽象メソッド (function)：

```
1 public interface MyFunction2 {
2   public double[][] function(double[] x, int nfunc);
3 }
```

1. L1: インタフェース MyFunction2.java の宣言である．
2. L2: 抽象メソッド function の宣言である．引数の x は関数値を求める座標値の配列である．nfunc は近似する関数の数である．

8.3　データの平滑化

データの補間の場合には，補間曲線はデータ点の上を通るけれども，データの平滑化の場合には，誤差を含んでばらつきのあるデータ $y_i, i = 1, \cdots, n$ から，滑らかなデータ曲線を作ることであるから，曲線は必ずしもデータ点の上を通るわけではない．最も単純な方法は，ある点 i の周りの数点 $N = 2m + 1$ の平均をとる

$$\frac{1}{N}\sum_{j=-m}^{m} y_{i+j} \tag{8.21}$$

であろう．これは，単純移動平均法といわれる．データが等間隔で与えられており，各データ点の周りで多項式近似が可能な場合には，周囲のいくつかの点の値に重みを掛けて平均をとるサビツキー・ゴーレイ法が使われることがある．この場合は，近似多項式が極めて局所的になるから，局所的な凹凸が残ることになる．

一方，B スプラインによるデータの平滑化の場合は，与えられるデータは等間隔でなくてもよい．データ点とは別に，節点を指定して B スプラインから平滑スプライン関数を構築するので，節点の数は比較的少数でもよい．そこで，一つの B スプラインが多くのデータ点を含む場合には，関数近似のように滑らかな関数が与えられる．しかし，極めて局所的な構造があっても，埋もれてしまうおそれはある．また，データに尖った点があることがわかっている場合には，その点に複数の節点を置くことで，尖りを表現することができる．

8.3.1　サビツキー・ゴーレイ法

この方法は，平滑化を行う測定データなどが等間隔で与えられており，そのデータ点の周りで多項式近似が可能な場合に使われる．その多項式の係数の数より多いデータ点を使って，最小二乗法でその多項式を近似することにより平滑化を施すことになる．そこで，この方法は多項式適合法ともいわれる．実際には，各 x 点について毎回多項式から平滑値を求めるのではなく，使用するデータ点の重み係数を使う形式にしておいて，それを使って平滑値を求めるようにしてある．表 8.2 は 3 次式を 5〜15 点を使って最小二乗法で決めたときの重み係数である．

この表を使うと，i 番目のデータの n 点による平滑化は次の式

$$y_i^{\mathrm{new}} = \frac{1}{N}\sum_{j=-(n-1)/2}^{(n-1)/2} w_j y_{i+j}^{\mathrm{old}} \tag{8.22}$$

となる．ここで，y_{i+j}^{old} は平滑化をする前のデータであり，y_i^{new} は平滑化により求められるデータである．

プログラム (Savitzky_Golay.java) のメソッド (savitzky_Golay) の一部とその説明は以下のとおりである．

表 8.2　3 次式によるサビツキー・ゴーレイ法の重み係数 w_j．x 点の近くの 5〜15 点のデータを使って最小二乗法で平滑化された 3 次式を決めたときの重み係数である．第 1 行の 0 は平滑化で修正されるデータ点を示しており，その列の数値は重みである．$\pm j$ の列は，その前後 j 番目のデータ点での重みである．第 1 列の n は使用する点の数であり，最後の列の N は規格化定数である．

n	0	±1	±2	±3	±4	±5	±6	±7	N
5	17	12	−3						35
7	7	6	3	−2					21
9	59	54	39	14	−21				231
11	89	84	69	44	9	−36			429
13	25	24	21	16	9	0	−11		143
15	167	162	147	122	87	42	−13	−78	1105

Savitzky_Golay.java のメソッド (savitzky_Golay) の一部：

```
 1  public double[] savitzky_Golay(int n, double[] y, int nmin, int nmax) {
 2    double[] z= new double[y.length];
 3    double[] r= new double[n];
 4    double[] rw;
 5    double d, sum;
 6    int min, max, i, j, k;
 7    double[] r5= {35, -3, 12, 17};
 8  中略
 9    if (n==5) {
10      rw= r5;
11    } else if (n==7) {
12  中略
13    int n2= n/2;
14    d= rw[0];
15    r[n2]= rw[n2+1]/d;
16    for (i=0; i<n2; i++) {
17      r[i]= rw[i+1]/d;
18      j= n-i-1;
19      r[j]= r[i];
20      j= nmin+i;
21      z[j]= y[j];
22      j= nmax-i;
23      z[j]= y[j];
24    }
25    min= n2+nmin;
26    max= nmax-n2;
27    for (i= min; i<=max; i++) {
28      k= i-n2 ;
29      sum= 0.0;
30      for (j=0; j<n; j++,k++) {
31        sum= sum+y[k]*r[j];
```

```
32        }
33        z[i]= sum ;
34      }
35      return z;
36    }
```

1. L1〜36: メソッド savitzky_Golay の定義である．引数 n は平滑化に使うデータの数，y は平滑化するデータの配列，nmin は配列 y のうち，平滑化するデータ範囲の最小値，nmax はその最大値である．戻り値は平滑後の配列である．
2. L2: 戻り値として返す平滑後の配列の宣言である．
3. L3: 計算に使うための重み係数の配列である．
4. L4: 実際に使うデータの配列の参照が代入される．
5. L7: n=5 の場合の係数の配列を作る．第 1 要素は規格化定数であり，その後に表 8.2 のデータが左から順に並んでいる．
6. L9, 10: 配列 rw に r5 の配列の参照を代入する．
7. L15, 17, 19: 係数を d で割って，本来の係数の値にしている．
8. L20〜23: 平滑化するデータの両端 n/2 点ずつは，この方法では平滑化できないから，元のデータの値をそのままコピーしておく．
9. L27〜34: 上記両端の内側で平滑化を繰り返す．
10. L28: i は平滑化するデータの添字番号，k は平滑化に使う添字の最小値である．
11. L29: データ値と重みの積和を作るためのゼロでの初期化．
12. L30〜32: n 回ループをまわし，積和を作る．
13. L33: 配列 z に平滑後の値を代入する．
14. L35: 配列 z を戻り値として返す．

図 8.4 は範囲 $[-1, 1]$ における等間隔での 100 点の x に対して，$y = x^2$ の値に $\sigma = 0.02$ の正規分布乱数を加算して作ったデータと，7 点 3 次式のサビツキー・ゴーレイ法を施して平滑化したデータである．一番下の + 印は平滑化前の元のデータであり，上に 0.25 シフトしてある ∘ 印は 1 回平滑化を施したデータ，0.5 シフトしてある
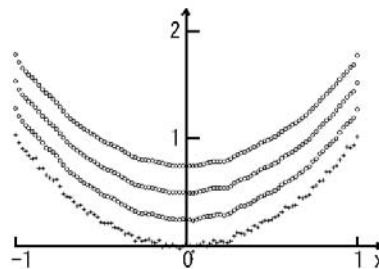
図 8.4　7 点 3 次式のサビツキー・ゴーレイ法による 2 次式データの平滑化．下から元のデータ，平滑化を 1 回，2 回，3 回施したデータであり，順に 0.25 ずつ上にずらして示されている．元のデータは関数 $y = x^2 + \epsilon$ であり，ϵ は $\sigma = 0.02$ の正規分布乱数である．データ点は $[-1, 1]$ の 100 分割で作ってある．

∘ 印は 2 回，0.75 シフトしてある ∘ 印は 3 回平滑化を繰り返して施したデータである．回数を増やすに従ってより平滑化が進んでいる．7 点を使う場合では，両端のそれぞれ 3 点については，式 (8.22) が使えない．平滑化には，多数の点の式を使うより，このように繰り返すほうが効率が良いようである．

図 8.5 は範囲 $[0,\ 10]$ における等間隔での 200 点の x に対して，$f(x) = \dfrac{1}{1+(x-3)^2}$ の値に $\sigma = 0.05$ の正規分布乱数を加算して作ったデータと，7 点 3 次式のサビツキー・ゴーレイ法を施して平滑化したデータである．一番下の $+$ 印は平滑化前の元のデータであり，上に 0.2 シフトしてある ∘ 印は 1 回平滑化を施したデータ，0.4 シフトしてある ∘ 印は 2 回，0.6 シフトしてある ∘ 印は 3 回平滑化を繰り返して施したデータである．回数を増やすに従ってより平滑化が進んでいるが，かなり細かい構造まで残っている．

8.3.2　B スプラインによる平滑化

再び B スプラインを使って，ここではデータの平滑化を行う．補間の場合，補間関数の式 (3.30) は

$$S(x) = \sum_{j=1}^{\nu+m} c_j N_{m,j}(x) \tag{8.23}$$

であり，データ点を通るから，連立方程式

$$S(x_i) = y_i \quad (i = 1, \cdots, n) \tag{8.24}$$

を解くことで係数 $c_i\,(i = 1, \cdots, n)$ を求めることができた．この補間の場合には，$n = \nu + m$ の関係がある．一方，平滑化の場合には，データ点の数 n は節点の数より
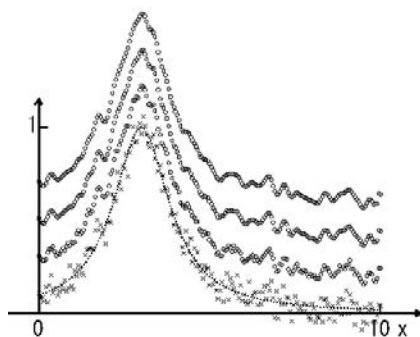
図 8.5　7 点 3 次式のサビツキー・ゴーレイ法によるピーク値のあるデータの平滑化．下から元のデータ，平滑化を 1 回，2 回，3 回施したデータであり，順に 0.2 ずつ上にずらして示されている．元のデータの関数は $f(x) = \dfrac{1}{1+(x-3)^2} + \epsilon$ であり，ϵ は $\sigma = 0.05$ の正規分布乱数である．データ点は $[0,\ 10]$ の 200 分割で作ってある．点線は $\epsilon = 0$ での式の曲線である．

十分に多いことが必要であり，データの平滑化は式 (8.23) に最小二乗法を適用した関数近似である．この場合の最小二乗法は，2 乗和

$$Q = \sum_{i=1}^{n} |S(x_i) - y_i|^2 \tag{8.25}$$

を係数 $c_i\,(i = 1, \cdots, \nu + m)$ で偏微分してゼロと置いてできる $\nu + m$ 元の連立方程式

$$A\mathbf{c} = \mathbf{b} \tag{8.26}$$

を解いて c_i を決定する方法である．この式の係数行列 A の要素は

$$a_{j,\,k} = \sum_{i}^{n} N_{m,\,j}(x_i) N_{m,\,k}(x_i) \tag{8.27}$$

であり，ベクトル $\mathbf{b}$ の要素は

$$b_j = \sum_{i}^{n} N_{m,\,j}(x_i) y_i \tag{8.28}$$

である．この場合も，式 (8.27) の和の中でゼロでない $N_{m,\,j}(x_i)$ は高々 m 個である．

プログラム (Bspl_Smooth.java) のメソッド (bspl_Smooth) と，その説明は以下のとおりである．

Bspl_Smooth.java のメソッド (bspl_Smooth)：

```
 1  public double bspl_Smooth(int nx, double[] xd, double[] yd,
 2      int nxi, double[] xi, int ior,
 3      int nn, double[] xx, double[] yy, double[] res) {
 4    int neq= nxi-ior;
 5    double[][] a= new double[neq][ior];
 6    double[] b= new double[neq];
 7    double[] c= new double[neq];
 8
 9    renritsu(nx,xd,yd, nxi,xi, ior, a,b);
10    c= cholesky(neq, ior, a,b);
11    double ans= keisan(nx,xd,yd, nxi,xi, ior, c,
12        nn,xx,yy, res);
13    return ans;
14  }
```

1. L1〜3〜14: メソッド (bspl_Smooth) の定義である．引数の nx は標本データの個数，xd は標本データの x 座標の配列，yd は標本データの y 座標の配列，nxi は付加節点を含めた節点の個数，xi は節点の x 座標の配列，ior は B スプラインの階数，nn は求めるデータの個数，xx は求めるデータの x 座標の配列，yy は求めるデータの値が戻る配列，res は求めるデータの値と真値との残差である．戻り値は標準偏差である．
2. L4: B スプラインの数 neq は，節点の数から階数を引いた値，nxi−ior である．
3. L5: a は連立方程式の係数行列であるが，各行でゼロでない値をもつ要素は階数以下であるから，第 2 添字の数は ior でよい．その代わり，連立方程式を解く次のメソッド choresky は複雑になっている．

4. L6, 7: b は連立方程式の右辺の定数のベクトルであり，c は解のベクトルで，スプライン多項式の係数である．
5. L9: a と b を求め，連立方程式を作成するメソッド renritsu を呼ぶ．
6. L10: 連立方程式をコレスキー法のメソッド choresky を呼んで解き，解を c に代入する．
7. L11, 12: 配列 xx に指定した x 値における平滑後の関数値，残差などの値を，メソッド keisan を呼んで計算する．戻り値として，標準偏差値を受け取る．
8. L13: 戻り値として，標準偏差値を返す．

図 8.6 は 3 次の B スプラインによるデータの平滑化である．関数は $f(x) = x^2 + \epsilon$ であり，$\sigma = 5.0$ の正規分布乱数 ϵ で，ばらつきをもたせてある．x の範囲 $[-1, 1]$ にデータは 101 個あり，節点の数は両端の付加節点 4 個ずつを含めて 17 個である．具体的には，$-1, -1, -1, -1,\ -0.8, -0.6, -0.4, -0.2,\ 0.0, 0.2, 0.4, 0.6, 0.8,\ 1, 1, 1, 1$ であり，付加節点以外は，等間隔にとってある．それらは x 軸上にバーで示されている．サビツキー・ゴーレイ法を使った平滑化の図 8.4 と比較して，遙かに滑らかになっている．しかし，実験データによっては，2 次式からはずれている細かい構造に意味がある場合もあり，ふさわしくないこともある．

図 8.7 は関数 $f(x) = \dfrac{1}{1+(x-3)^2} + \epsilon$ のデータに 3 次の B スプラインによる平滑化を施した例である．$\sigma = 0.05$ の正規分布乱数 ϵ で，ばらつきをもたせてある．データ数は 201 個あり，節点の数は両端の付加節点 4 個ずつを含めて 17 個である．具体的には（0, 0, 0, 0, 0.75, 1.25, 2, 2.5, 3.75, 5, 6.25, 7.5, 8.75, 10, 10, 10, 10）の 17 点であり，x 軸上にバーで示されている．ピークの左側では右側より多くの節点を使っているが，変化のより大きな左側で節点の数を増やさないと平滑曲線に大きな振動が生じてしまう．

発散点や鋭いピーク値などの特異点がある場合には，その特異点を 3 重の節点とすることによって，表現することができる．図 8.8 の関数は $f(x) = \dfrac{1}{0.1+(x-3)^2} + \epsilon$ であり，関数のピークとなる $x = 3$ で $f(3) = 10.0$ となるように，図 8.7 の分母の定数
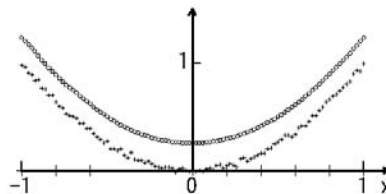
図 8.6　3 次の B スプラインによる 2 次式データの平滑化．関数は $f(x) = x^2 + \epsilon$ である．ϵ は $\sigma = 5.0$ の正規分布乱数である．データ点は $[-1, 1]$ に等分割で 101 個作っている．節点は両端の付加節点を含め 17 点であり，x 軸上にバーで示されている．× 印がデータであり，○ 印が平滑化を施した値で，0.25 上にシフトしてある．（Z08_06_Bspl_Smooth_2ji.java）
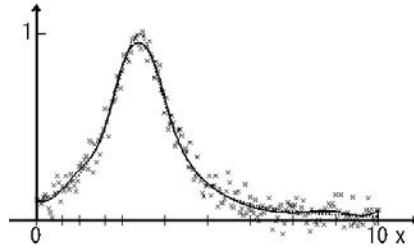
図 8.7　3 次の B スプラインによるピーク値のあるデータの平滑化．関数は $x = 3$ にピークをもつ $f(x) = \dfrac{1}{1+(x-3)^2} + \epsilon$ であり，ϵ は $\sigma = 0.05$ の正規分布乱数である．データ点は $[0, 10]$ で 201 点ある．両端の付加節点を含めて 17 点の節点は，x 軸上にバーで示されている．× 印がデータ，実線が平滑曲線，点線が $\epsilon = 0$ での式の曲線である．（Z08_07_Bspl_Smooth.java）
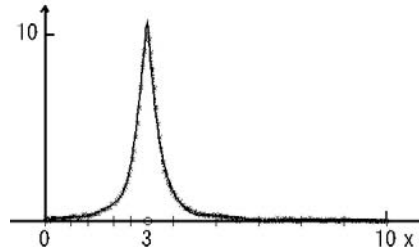
図 8.8　3 次の B スプラインによる鋭いピーク値のあるデータの平滑化．関数は $x = 3$ に鋭いピークをもつ $f(x) = \dfrac{1}{0.1+(x-3)^2} + \epsilon$ であり，ϵ は $\sigma = 0.05$ の正規分布乱数である．データ点は $[0, 10]$ で 201 点ある．両端の付加節点を含めて 20 点の節点は，x 軸上にバーで示されており，3 重節点である 3 は ∘ 印で示されている．× 印がデータ，実線が平滑曲線である．（Z08_08_Bspl_Smooth_peak.java）

を 1 から 0.1 に変えてある．このピークを表現するために，先の節点 17 個のほかに，ピーク値となる $x = 3$ の値を 3 個追加して，(0, 0, 0, 0, 0.75, 1.25, 2, 2.5, 3, 3, 3, 3.75, 5, 6.25, 7.5, 8.75, 10, 10, 10, 10) の 20 点としてある．3 重節点は x 軸上の 3 に ∘ 印で示されている．

スプライン関数によって平滑曲線を作るとき，データは多くなければならないが，節点はうまくとればかなり少数でよいことが多い．図を見ながら，試行錯誤で節点を追加することもできるが，すべてのデータ点において現在の平滑曲線と適当な閾値以上食い違っている点があったら，その付近に節点を追加していく，逐次分割法などのアルゴリズムがあることを付け加えておく．

8.3.3　フーリエ変換を使う平滑化

第 10 章で扱うフーリエ変換の一種で効率の良い FFT を使ってデータの平滑化を行う．平滑化に使う関数に，ガウス関数やローレンツ関数を用いれば，畳み込みになる．これは，サビツキー・ゴーレイ法の重み係数の代わりに，ガウス関数やローレンツ関数の値を使うことと同等である．ガウス関数やローレンツ関数のパラメータ σ や Γ をより大きくとれば，より遠くのデータを取り入れることになり，より小さくとればより近くのデータしか取り入れないことになる．B スプラインのように，節点の選び方のような難しさはなく，パラメータの値を選ぶだけで平滑化の程度を制御することができるので簡単である．また，パラメータの値の選び方にもよるが，サビツキー・ゴーレイ法より広域的な平滑化をすることもできる．

図 8.9 は，図 8.5 や図 8.7 と同様なデータを，ガウス関数を使った FFT を用いて平滑化したものである．$\times$ 点で示されている関数は $f(x) = \dfrac{1}{1+(x-3)^2} + \epsilon$ であり，ϵ は $\sigma = 0.05$ の正規分布乱数である．重なっている曲線は，$\epsilon = 0$ の真の関数である．左の曲線は，下から $\sigma = 0.1, 0.2, 0.4, 0.8$ のガウス曲線であり，対応した右の図は，それぞれのガウス関数を使って FFT の畳み込みで平滑化したものである．$\sigma = 0.1$ の場合は，ピーク値は低くなっていないが，かなり，ばらつきが残っている．$\sigma = 0.8$ の場合は，ばらつきは消えているが，ピーク値はかなり低くなっており，$\sigma = 0.4$ 程度が良さそうである．

データ点は 201 点あるが，FFT に乗せるためには 512 点，あるいは 1,024 点に拡張する必要がある．ここでは，ピークの中心の $x = 3$ が 1,024 点の中央に来るようにデータをコピーし，データが不足する両方の外側にはゼロを代入しておく．このまま
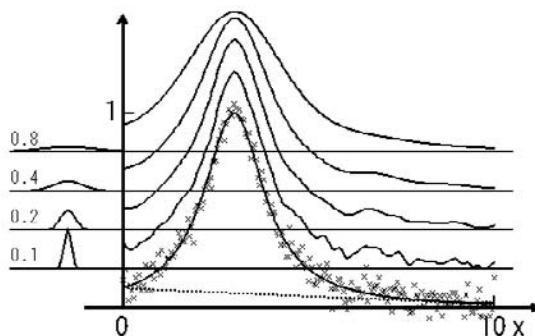
図 8.9　FFT を用いた平滑化．関数は $f(x) = \dfrac{1}{1+(x-3)^2} + \epsilon$ である．ϵ は $\sigma = 0.05$ の正規分布乱数である．データ点は $[0, 10]$ で 201 点あり，$\times$ 点で示されている．重なっている曲線は，$\epsilon = 0$ の真の関数である．点線は，フーリエ変換で意味のない振動が入らないように，グラフの両端で値がゼロに近づくように引き去る下駄の関数である．下から 2 番目の実曲線は，左に示されている $\sigma = 0.1$ のガウス関数を使って FFT の畳み込みで平滑化したものであり，上方へ 20 ずつシフトしてある．下からさらに，$\sigma = 0.2, 0.4, 0.8$ の場合の曲線である．

ではデータの両端で段差が生じ，フーリエ変換で無意味な振動が入ってくるので，図の点線で示してある関数を下駄として引き去り，段差をなくしている．そこで，平滑化の後でその下駄を履かせている．プログラムの詳細は後にゆずる．

演習問題

8.1　次の表はある果物の収穫量 y の，5 月の平均気温 x に対する数値である．x と y の間に 2 次式の関係 $y = ax^2 + bx + c$ があると仮定して，最小二乗法で係数 a, b, c を決めなさい．

x	20.0	20.5	21.0	21.5	22.0	22.5	23.0
y	12.0	12.3	14.4	15.1	18.9	20.3	21.7
x	23.5	24.0	24.5	25.0	25.5	26.0	
x	22.0	21.1	20.4	18.1	16.9	13.0	

答： $a = -0.989$, $b = 46.217$, $c = -518.956$

解答例： ソースプログラム Q08_01_Z08_02_SJH_2jishiki.java

8.2　3 次の B スプラインを使った平滑化のプログラム Z08_07_Bspl_Smooth.java の節点を変えて，平滑化の様子を調べなさい．

第 9 章

固有値問題

固有値問題は振動，波動現象などの解析に必須なことであり，構造物の振動や，量子物理学の波動関数を扱う分野では最も重要な数値計算法の一つである．一方，2 次曲線の主軸を求める主軸問題は最も簡単な固有値問題である．

9.1 主軸問題

中心は原点にあるが，軸が傾いている楕円を扱うことにする．基本となる座標軸を x, y とし，傾いた楕円の方程式を

$$ax^2 + 2fxy + by^2 = 1 \tag{9.1}$$

とする．原点を中心に角 θ だけ右 (時計回り) に回転した座標軸 x', y' がこの楕円の主軸となっているならば，この楕円の方程式は

$$a'x'^2 + b'y'^2 = 1 \tag{9.2}$$

である[1]．両式を行列を使って表すと，それぞれ

$$(x,\, y)\begin{pmatrix} a, & f \\ f, & b \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = 1 \tag{9.3}$$

と

$$(x',\, y')\begin{pmatrix} a', & 0 \\ 0, & b' \end{pmatrix}\begin{pmatrix} x' \\ y' \end{pmatrix} = 1 \tag{9.4}$$

である．また，両座標系での座標値（x, y）と（x', y'）の関係は図 9.1 からわかるように

$$\begin{aligned} x' &= x\cos\theta - y\sin\theta \\ y' &= x\sin\theta + y\cos\theta \end{aligned} \tag{9.5}$$

である．これを，行列を使って表すと

---

1. 楕円の方程式が式 (9.2) で表せるとき，その座標軸を主軸という．
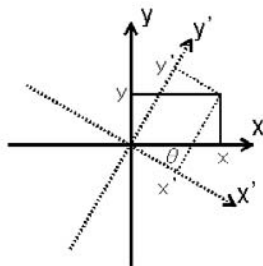
図 9.1　x, y 座標と右回りに θ だけ回転している x', y' 座標.

$$(x',\,y') = (x,\,y)\begin{pmatrix} \cos\theta, & \sin\theta \\ -\sin\theta, & \cos\theta \end{pmatrix},\ \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta, & -\sin\theta \\ \sin\theta, & \cos\theta \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} \tag{9.6}$$

あるいは,

$$(x,\,y) = (x',\,y')\begin{pmatrix} \cos\theta, & -\sin\theta \\ \sin\theta, & \cos\theta \end{pmatrix},\ \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \cos\theta, & \sin\theta \\ -\sin\theta, & \cos\theta \end{pmatrix}\begin{pmatrix} x' \\ y' \end{pmatrix} \tag{9.7}$$

である．式 (9.7) の関係を式 (9.3) に代入すると,

$$(x',\,y')\begin{pmatrix} \cos\theta, & -\sin\theta \\ \sin\theta, & \cos\theta \end{pmatrix}\begin{pmatrix} a, & f \\ f, & b \end{pmatrix}\begin{pmatrix} \cos\theta, & \sin\theta \\ -\sin\theta, & \cos\theta \end{pmatrix}\begin{pmatrix} x' \\ y' \end{pmatrix} = 1 \tag{9.8}$$

となる．これを式 (9.4) の形式にまとめると,

$$(x',\,y')\begin{pmatrix} a\cos^2\theta - f\sin 2\theta + b\sin^2\theta, & \frac{1}{2}(a-b)\sin 2\theta + f\cos 2\theta \\ \frac{1}{2}(a-b)\sin 2\theta + f\cos 2\theta, & a\sin^2\theta + f\sin 2\theta + b\cos^2\theta \end{pmatrix}\begin{pmatrix} x' \\ y' \end{pmatrix} = 1 \tag{9.9}$$

となる．式 (9.4) と比較すると，式 (9.9) の行列の非対角要素がゼロとなるためには,

$$\frac{1}{2}(a-b)\sin 2\theta + f\cos 2\theta = 0 \tag{9.10}$$

でなければならない．これより,

$$\begin{array}{ll} a \neq b \text{ の場合} & \tan 2\theta = \dfrac{-2f}{a-b} \\ a = b,\ f \neq 0 \text{ の場合} & \cos 2\theta = 0 \\ a = b,\ f = 0 \text{ の場合} & \theta = \text{不定} \end{array} \tag{9.11}$$

であり，さらに

$$\begin{array}{ll} a \neq b \text{ の場合} & \theta = \dfrac{1}{2}\tan^{-1}\dfrac{-2f}{a-b} \\ a = b,\ f \neq 0 \text{ の場合} & \theta = 45^\circ + n \times 90^\circ\ (n \text{ は整数}) \\ a = b,\ f = 0 \text{ の場合} & \theta = \text{不定} \end{array} \tag{9.12}$$

である．この $a = b$, $f \neq 0$ の場合は，式が x と y について対称になっているから，主軸は $\pm 45°$ の方向である．また，$a = b$, $f = 0$ の場合は，方程式は $ax^2 + ay^2 = 1$ であるから円であり，主軸の方向は不定である．さらに式 (9.9) の行列の対角要素を式 (9.4) と比較すると，

$$
\begin{aligned}
a' &= a\cos^2\theta - f\sin 2\theta + b\sin^2\theta \\
b' &= a\sin^2\theta + f\sin 2\theta + b\cos^2\theta
\end{aligned}
\tag{9.13}
$$

が決まる．このときの主軸の半径は，x' 軸と y' 軸とで，それぞれ $\dfrac{1}{\sqrt{a'}}$ と $\dfrac{1}{\sqrt{b'}}$ である．

図 9.2 は $3x^2 + 2\sqrt{3}xy + 5y^2 = 1$ のグラフと，点線は後で解説するヤコビ法で求めた主軸 x', y' である．長軸の半径は 0.70710，短軸の半径は 0.40825 である．
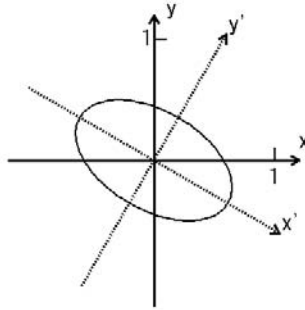
図 9.2　$3x^2 + 2\sqrt{3}xy + 5y^2 = 1$ の楕円と主軸 x', y' (点線)．長軸の半径は 0.70710，短軸の半径は 0.40825 である．(Z09_02_JacobiDaen)

9.2　固有値と固有ベクトル

2 次元の主軸問題は上に述べたように，非対角要素がゼロになるようにして，簡単に解くことができる．この方法は後で述べるヤコビ法を 2 次元行列に適用したことになっている．しかし，3 次元以上になると，非対角要素が 3 組あるので，一度ゼロにした非対角要素も，別の非対角要素をゼロにする変換操作でゼロでなくなってしまう．そこで，3 次元以上の場合には，計算機に頼る必要がある．次元数が小さいときに効率が良くて簡便な直接法やヤコビ法，絶対値の大きい固有値を数個求めるのに適した累乗法，行列の次元数が大きいときに効率が良い本格的な QR 法などがある．まず，主軸問題と固有値問題の関係を説明する．

9.2.1　固有値問題

ここで，式 (9.8) と式 (9.4) を比較すると，

$$\begin{pmatrix} \cos\theta, & -\sin\theta \\ \sin\theta, & \cos\theta \end{pmatrix} \begin{pmatrix} a, & f \\ f, & b \end{pmatrix} \begin{pmatrix} \cos\theta, & \sin\theta \\ -\sin\theta, & \cos\theta \end{pmatrix} = \begin{pmatrix} a', & 0 \\ 0, & b' \end{pmatrix} \tag{9.14}$$

となる．ここで，

$$A = \begin{pmatrix} a, & f \\ f, & b \end{pmatrix}, \quad S = \begin{pmatrix} \cos\theta, & \sin\theta \\ -\sin\theta, & \cos\theta \end{pmatrix}, \quad D = \begin{pmatrix} a', & 0 \\ 0, & b' \end{pmatrix} \tag{9.15}$$

と置くと，式 (9.14) は直交変換[2]の式

$$S^T AS = D \tag{9.16}$$

となる．S^T は S の転置行列である．行列 S のように

$$\sum_i s_{ij}s_{ik} = \sum_i s_{ji}s_{ki} = \delta_{jk} = \begin{cases} 1 & (j = k) \\ 0 & (j \neq k) \end{cases} \tag{9.17}$$

が成り立つ行列のことを直交行列という．ただし，s_{ij} などは行列 S の要素であり，δ_{jk} はクロネッカーのデルタである．

式 (9.16) の両辺に左から S を掛けると，

$$AS = SD \tag{9.18}$$

となる．これは，

$$\begin{aligned} A\begin{pmatrix} \cos\theta \\ -\sin\theta \end{pmatrix} &= a'\begin{pmatrix} \cos\theta \\ -\sin\theta \end{pmatrix} \\ A\begin{pmatrix} \sin\theta \\ \cos\theta \end{pmatrix} &= b'\begin{pmatrix} \sin\theta \\ \cos\theta \end{pmatrix} \end{aligned} \tag{9.19}$$

のように書きなおすことができる．式 (9.19) で，a' と b' は行列 A の固有値，$\mathbf{s}_1 = \begin{pmatrix} \cos\theta \\ -\sin\theta \end{pmatrix}$ と $\mathbf{s}_2 = \begin{pmatrix} \sin\theta \\ \cos\theta \end{pmatrix}$ は，それぞれ，固有値 a' と b' に属する固有ベクトルといわれる．先の楕円の主軸問題との兼ね合いでいうと，固有値 a' と b' は 2 本の主軸の半径 $\dfrac{1}{\sqrt{a'}}$ と $\dfrac{1}{\sqrt{b'}}$ の逆数の 2 乗であり，固有ベクトル $\mathbf{s}_1$ と $\mathbf{s}_2$ は各主軸方向の単位ベクトルである．一方，$S = (\mathbf{s}_1, \mathbf{s}_2)$ であり，$\mathbf{s}_1$ と $\mathbf{s}_2$ は行列 S の列ベクトルである．S は直交行列であるから，各列の大きさは 1 であり，異なる 2 列の内積はゼロである．そこで，固有ベクトルには直交規格関係が成り立つ．

一般的に n 次元の実対称行列 (要素が複素数の場合にはエルミート行列) A の固有値，固有ベクトルを求める場合には，式 (9.18) の S は n 次元の直交行列 (要素が複

[2]. ある行列を直交行列とその転置行列で挟んで変換することを直交変換という．

素数の場合にはユニタリー行列)，D は n 次元の実対角行列である．式 (9.19) は一般的に，

$$A\mathbf{s}_i = d_i\mathbf{s}_i \quad (i = 1, \cdots, n) \tag{9.20}$$

と書くことができる．これをさらに書き換えると，式 (9.20) の n 個の式に対して共通な式

$$\begin{pmatrix} a_{11}-d & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22}-d & \cdots & a_{2n} \\ & & \cdots & \\ a_{n1} & a_{n2} & \cdots & a_{nn}-d \end{pmatrix} \begin{pmatrix} s_1 \\ s_2 \\ \cdot \\ s_n \end{pmatrix} = 0 \tag{9.21}$$

となり，これは n 元連立同次方程式である．この連立方程式が $\mathbf{s} = \mathbf{0}$ 以外の解をもつためには，d が

$$\begin{vmatrix} a_{11}-d & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22}-d & \cdots & a_{2n} \\ & & \cdots & \\ a_{n1} & a_{n2} & \cdots & a_{nn}-d \end{vmatrix} = 0 \tag{9.22}$$

の根でなければならない．この式 (9.22) は行列 A の固有方程式といわれる．

この固有方程式が対角行列か，下三角行列あるいは上三角行列であれば，式 (9.22) は

$$(a_{11}-d)(a_{22}-d)\cdots(a_{nn}-d) = 0 \tag{9.23}$$

となるので，固有値は

$$d = a_{i,i} \quad (i = 0, \cdots, n) \tag{9.24}$$

である．そこで，固有値問題を解くということは，行列 A に変換を施して，対称行列の場合には対角行列に，非対称行列の場合には下三角行列あるいは上三角行列にすることであるともいえる．ただし，変換によって固有値が変わらないようにしなければならない．そこで，先の主軸問題で見たように，楕円 (一般的には n 次元空間での 2 次形式) の形を変えないように，直交変換の式 (9.16) を使うことになる．以下で述べる，ヤコビ法や QR 法ではこの考えを使っている．

9.2.2　直接法

固有方程式 (9.22) の解である固有値 d を 2 分法などを使って求め，各固有値ごとにその値を式 (9.21) に代入して，固有ベクトルを求めることができる．まず，固有値の一つの d を式 (9.21) に代入する．次に，s_i のどれか一つ s_p の値を 1 として式 (9.21) に代入し，その列を右辺に移項すると

$$\left.\begin{array}{llllll}(a_{11}-d)s_1 & +a_{12}s_2 & +\cdots & +a_{1n}s_n & = & -a_{1p} \\ a_{21}s_1 & +(a_{22}-d)s_2 & +\cdots & +a_{2n}s_n & = & -a_{2p} \\ & & \cdots & & & \\ a_{p1}s_1 & +a_{p2}s_2 & +\cdots & +(a_{pn}-d)s_n & = & -(a_{pp}-d) \\ & & \cdots & & & \\ a_{n1}s_1 & +a_{n2}s_2 & +\cdots & +(a_{nn}-d)s_n & = & -a_{np}\end{array}\right\} \quad (9.25)$$

となる．これは，$(n-1)$ 元連立 1 次方程式であるが，式の数は n 個あるので，一つの式を捨てることになる．この場合，$(n-1)$ 個の方程式が 1 次独立でなければならないから，捨てる方程式をどれにするか気を付けなければならないこともある．この連立方程式を解いて，残りの $(n-1)$ 個の s_i を求め，最後にベクトル $\mathbf{s}$ を規格化しなおせば，固有ベクトルとなる．この方法は効率が良くないが，行列 A の要素が固有値に依存するような拡張された固有方程式の場合には，この方法に頼らざるを得ない．

9.2.3　ヤコビ法

ここでは，比較的簡単なヤコビ法を説明する．A が 2 次元対称行列の場合の固有値問題はすでに述べたとおりであり，式 (9.14) を満たす θ を求めればよく，それは式 (9.12) である．n 次元の場合には，n 次元の対称行列 A に対する式 (9.16) の S を求めなければならない．まず，適当な直交行列 S_1 により，A に直交変換 $S_1^T AS_1$ を施し，どれか一つの非対角要素を 0 にする．次に，また適当な直交行列 S_2 により，直交変換 $S_2^T S_1^T AS_1S_2$ を施し，別の一つの非対角要素を 0 にする．その際，先に 0 にした非対角要素が 0 でなくなることもあるが，元の値よりは 0 に近くなっている．そこで，すべての非対角要素が十分小さくなるまで直交変換を m 回繰り返せば，式 (9.16) は，

$$\begin{aligned} S &= S_1S_2\cdots S_m \\ S^TAS &\cong D \end{aligned} \quad (9.26)$$

とすることができる．ここで，対称行列 A の A_{pq} 要素を消去する直交変換の直交行列を扱う．それは

$$S_i = \begin{pmatrix} 1 & 0 & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & 1 & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & 0 & \cos\theta & \cdots & \sin\theta & \cdots & 0 \\ 0 & \cdots & \cdots & 1 & \cdots & \cdots & 0 \\ 0 & \cdots & -\sin\theta & \cdots & \cos\theta & 0 & 0 \\ 0 & \cdots & \cdots & \cdots & 0 & 1 & 0 \\ 0 & \cdots & \cdots & \cdots & 0 & 0 & 1 \end{pmatrix} \begin{array}{l} \\ \\ \cdots > \quad p\,行 \\ \\ \cdots > \quad q\,行 \\ \\ \\ \end{array} \quad (9.27)$$

である．成分で書くと

$$\begin{aligned} & s_{pp} = \cos\theta, \quad s_{pq} = \sin\theta \\ & s_{qp} = -\sin\theta, \quad s_{qq} = \cos\theta \\ & s_{ii} = 1, \quad s_{iq} = s_{qi} = s_{ip} = s_{pi} = s_{ik} = 0 \quad ただし\, i \neq p, q, \quad k \neq p, q \end{aligned} \quad (9.28)$$

である．直交変換の結果の $B = S^T AS$ を調べると，

$$\left.\begin{aligned} B_{pk} &= A_{pk}\cos\theta - A_{qk}\sin\theta \\ B_{qk} &= A_{pk}\sin\theta + A_{qk}\cos\theta \\ B_{kp} &= A_{kp}\cos\theta - A_{kq}\sin\theta \\ B_{kq} &= A_{kp}\sin\theta + A_{kq}\cos\theta \\ B_{ik} &= A_{ik} \end{aligned}\right\} \quad (i,\, k \neq p,\, q)$$
$$\begin{aligned} B_{pp} &= A_{pp}\cos^2\theta - A_{pq}\sin 2\theta + A_{qq}\sin^2\theta \\ B_{qq} &= A_{pp}\sin^2\theta + A_{pq}\sin 2\theta + A_{qq}\cos^2\theta \\ B_{qp} &= B_{pq} = \frac{1}{2}(A_{pp} - A_{qq})\sin 2\theta + A_{pq}\cos 2\theta \end{aligned} \tag{9.29}$$

である．B_{pq} を消去するために，式 (9.11) と同じく

$$\tan 2\theta = \frac{-2A_{pq}}{A_{pp} - A_{qq}} \tag{9.30}$$

が成り立たねばならず，これから θ の値が決まる．しかし，実際には $\tan 2\theta$ と $\sin\theta$, $\cos\theta$ との間にある次の関係

$$\begin{aligned} \lambda &= -A_{pq} \\ \mu &= \frac{1}{2}(A_{pp} - A_{qq}) \\ \tan 2\theta &= \frac{\lambda}{\mu} \\ \omega &= \operatorname{sgn}(\mu)\frac{\lambda}{\sqrt{\lambda^2 + \mu^2}} \\ \sin\theta &= \frac{\omega}{\sqrt{2(1 + \sqrt{1 - \omega^2})}} \\ \cos\theta &= \sqrt{1 - \sin^2\theta} \end{aligned} \tag{9.31}$$

が使えるから，θ を求める必要はない．関数 sgn( ) は引数の正負の符号を返す関数である．

行列 A の非対角要素を消去していく順序はいろいろ考えられ，

(1) 0 でない非対角要素を端から順番に消去していく．
(2) 絶対値の大きな非対角要素から順番に消去していく．
(3) 絶対値がある値 t 以上の非対角要素を端から順番に消去し，t を次第に小さくしていく．

等がある．実例のプログラムでは (3) を使っている．

図 9.3 には 3 次元楕円体面 $2x^2 + y^2 + 3z^2 + 0.6xy + 0.8yz + zx = 1$ と，点線で座標軸 x, y, z が，実線でヤコビ法で求められ主軸 x', y', z' が示されている．

対称実行列固有値問題のプログラム (Jacobi.java) のメソッド (jacobi) と，その説明は以下のとおりである．
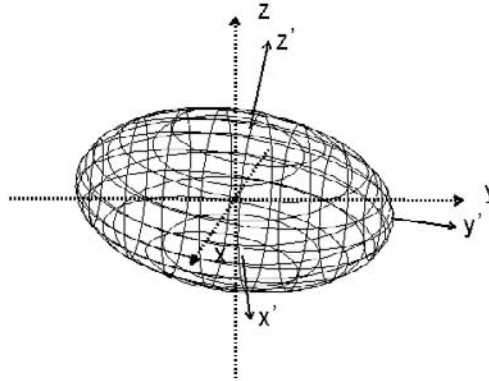
図 9.3　3 次元楕円体面と主軸．3 次元楕円体面は $2x^2 + y^2 + 3z^2 + 0.6xy + 0.8yz + zx = 1$ である．座標軸 x, y, z が点線で，ヤコビ法で求められた主軸 x', y', z' が実線で示されている．(Z09_03_JacobiDaenMen.java)

Jacobi.java のメソッド (jacobi)：

```
 1   public void jacobi(double[][] a, double[][] s, double[] d,
 2     double eps, boolean vector)  {
 3     int n= a.length;
 4     double onm, fnu, t, mu, lam, omg, pp, qq, qp;
 5     double ap, aq, sp, sq;
 6     double x, sin1, sin2, cos1, cos2;
 7     boolean ind ;
 8 //  S を単位ベクトルにする
 9     if (vector) {
10       for (int i=0; i<n; i++) {
11         s[i][i]= 1.0 ;
12         for (int j= i+1; j<n; j++ ) {
13           s[i][j]= 0.0;
14           s[j][i]= 0.0;
15         }
16       }
17     }
18 //  非対角ノルム sqrt{sum(i ne j) aij^2} の計算
19     onm= 0.0 ;
20     for (int i=0; i<n-1; i++) {
21       for (int j=i+1; j<n; j++) {
22         onm= onm+(a[i][j]*a[i][j]);
23       }
24     }
25     onm= Math.sqrt(onm)/n;
26 //  最後の閾値 fnu
27     fnu= onm*eps;
28     t= onm;
29     do {
```

```
30 //  最初の閾値 t は非対角ノルムの 1/n ずつ小さくし，最後は 0.99 fnu
31        t= t/n;
32        if (t < fnu) t= fnu*0.99;
33        do {
34          ind= false ;
35          for (int q=1; q<n; q++) {
36            for (int p=0; p<=q-1; p++) {   // 上三角で処理
37              if (Math.abs(a[q][p]) > t) { // 大きい要素あり
38                ind= true ;
39 //  sin θ, cos θ の計算
40                mu= 0.5*(a[p][p]-a[q][q]);
41                lam= -a[q][p] ;
42                omg= lam/Math.sqrt(lam*lam+mu*mu);
43                if (mu < 0.0) {
44                 omg= -omg;
45                }
46                sin1= omg/Math.sqrt(2*(1+Math.sqrt(1-omg*omg)));
47                cos1= Math.sqrt(1-sin1*sin1);
48                sin2= sin1*sin1;
49                cos2= cos1*cos1;
50                pp= a[p][p];
51                qq= a[q][q];
52                qp= 2.0*a[q][p]*sin1*cos1;
53 //  a の p,q 行，p,q 列の変換
54                for (int k=0; k<n; k++) {
55                  if (k==p || k==q) continue;
56                  ap= a[p][k] ;
57                  aq= a[q][k] ;
58                  a[p][k]= ap*cos1-aq*sin1;
59                  a[q][k]= ap*sin1+aq*cos1;
60                  a[k][p]= a[p][k];
61                  a[k][q]= a[q][k];
62                }
63 //  a の枢軸の計算
64                a[p][p]= pp*cos2-qp+qq*sin2;
65                a[q][q]= pp*sin2+qp+qq*cos2;
66                a[q][p]= a[p][q]= 0.0;
67 //  s の計算
68                if (vector) {
69                  for (int k=0; k<n; k++) {
70                    sp= s[k][p] ;
71                    sq= s[k][q] ;
72                    s[k][p]= sp*cos1-sq*sin1;
73                    s[k][q]= sp*sin1+sq*cos1;
74                   }
75                }
76              }
77            }
78          }
79        } while (ind);
80      } while (t >= fnu);
```

```
81 //  a の対角要素が固有値になっているから d に代入する
82     for (int i=0; i<n; i++) {
83       d[i]= a[i][i];
84     }
85   }
```

1. L1, 2〜85: メソッド jacobi の定義である．固有値問題の式は $AS = SD$ であり，引数の a は行列 A，s は固有ベクトルが縦ベクトルとなっている行列 S，d は固有値のベクトル $\mathbf{d}$ である．eps は行列 A を対角化していくときの非対角要素の相対精度の閾値，vector は固有ベクトルも求めるときは true を指定し，求めないときは false を指定する boolean である．
2. L9〜17: 引数 vector が true なら，固有ベクトルを求めていく行列 S を単位行列とする．
3. L19〜25: 非対角要素の閾値を決めるために，非対角要素全体のノルム (大きさ) を使って onm を求める．
4. L27: 最終的な閾値の値を fnu に指定する．
5. L28: 途中の閾値 t の初期値として，先のノルム onm を指定する．
6. L29〜80: 閾値 t を 1/n ずつ小さくして，fnu より小さくなるまで繰り返すループである．
7. L32: 最後の t が小さくなりすぎないようにしている．
8. L33〜79: すべての非対角要素を閾値 t より小さくするループである．
9. L34: 今回のループの処理で，すべての非対角要素が閾値 t より小さくなっていたらループを抜けるためのフラグ ind を false にしておく．
10. L35, 36〜77, 78: 上三角行列部分でまわすためのループである．
11. L37〜76: 閾値より大きい非対角要素が見つかったら処理をする．
12. L38: フラグ ind を true にする．
13. L40〜47: 式 (9.31) による $\sin\theta$, $\cos\theta$ の計算．
14. L48〜66: 式 (9.29) による変換を行う．
15. L68〜75: 引数 vector が true なら，固有ベクトルについても変換を行う．
16. L76: 閾値より大きい非対角要素が見つけたときの処理の終了．
17. L77, 78: 上三角行列部分でまわすためのループの終端．
18. L79: すべての非対角要素が閾値 t より小さくなっていたらループを抜ける．
19. L80: すべての非対角要素が最終的な閾値の fnu より小さくなっていたらループを抜ける．
20. L82〜84: a の対角要素が固有値になっているから d に代入する．

9.2.4 累乗法

次元の大きな行列を扱うが，絶対値が大きな固有値を少しだけ必要とする場合には，累乗法 (power method) のほうが有効な場合がある．A を n 次元実対称行列とし，まだ未知であるが n 個の固有値は絶対値の大きい順 $|d_1| \geq \cdots \geq |d_n|$ の $d_1, \cdots, d_n$ であり，対応する固有ベクトルは $\mathbf{s}_1, \cdots, \mathbf{s}_n$ であるとする．

あるベクトル $\mathbf{s}^{(\mathbf{0})} = \displaystyle\sum_{i=1}^{n} \alpha_i \mathbf{s}_i$ を選び，これに左から A を掛けると，

$$\mathbf{s}^{(1)} = A\mathbf{s}^{(0)} = \sum_{i=1}^{n} \alpha_i A\mathbf{s}_i = \sum_{i=1}^{n} \alpha_i d_i \mathbf{s}_i \tag{9.32}$$

である．左から A を繰り返し掛けると，

$$\mathbf{s}^{(p)} = \sum_{i=0}^{n} \alpha_i d_i^p \mathbf{s}_i = d_1^p \left(\alpha_1 \mathbf{s}_1 + \sum_{i=2}^{n} \alpha_i \left(\frac{d_i}{d_1} \right)^p \mathbf{s}_i \right) \tag{9.33}$$

となり，p が大きくなると，

$$\mathbf{s}^{(p)} = \alpha_1 d_1^p \mathbf{s}_1 \tag{9.34}$$

に近づく．このことは，絶対値が最大の A の固有値に対応する固有ベクトルの方向に収束していくことを示している．繰り返しのたびにベクトル $\mathbf{s}^{(p)}$ を規格化しておけば，収束したときには，

$$A\mathbf{s}^{(p)} = d\mathbf{s}^{(p)} \tag{9.35}$$

が成り立っているから，d は固有値であり，そのときの $\mathbf{s}$ は固有ベクトルとなっている．このように，A を繰り返し左から掛けていくので，この方法を累乗法という．

その固有値と固有ベクトルを，それぞれ d_1，$\mathbf{s_1}$ として，

$$A_2 = A - d_1 \mathbf{s_1} \mathbf{s_1}^T \tag{9.36}$$

に累乗法を適用すると，

$$\begin{aligned} A_2 \mathbf{s_1} &= A\mathbf{s_1} - d_1 \mathbf{s_1} \mathbf{s_1}^T \mathbf{s_1} = d_1 \mathbf{s_1} - d_1 \mathbf{s_1} \times 1 = 0 \\ A_2 \mathbf{s_i} &= d_i \mathbf{s_i} - d_1 \mathbf{s_1} \mathbf{s_1}^T \mathbf{s_i} = d_i \mathbf{s_i} - d_1 \mathbf{s_1} \times 0 = d_i \mathbf{s_i} \end{aligned} \tag{9.37}$$

であるから，2 番目に絶対値の大きな固有値 d_2 と固有ベクトル $\mathbf{s_2}$ が求められる．$\mathbf{s_i}^T$ は $\mathbf{s_i}$ の転置ベクトルであり，列ベクトルと行ベクトルの積 $\mathbf{s_i}\mathbf{s_j}^T$ は $(n \times n)$ の行列になる．また，$\mathbf{s_i}$ は規格直交化された固有ベクトルであるから，式 (9.17) の直交規格関係が成り立ち，それを使っている．3 番目以降の固有値と固有ベクトルも同様にして求められる．しかし，$\left| \dfrac{d_2}{d_1} \right|$ の値が小さい場合などでは収束が遅く，また，すべての対角要素が小さい場合には収束しないこともある．そのような場合には，行列の対角要素を一定の値 c だけずらして

$$A' = A + cI \tag{9.38}$$

とすれば，速やかに収束させられることがある．その場合の真の固有値は $d_i = d_i' - c$ である．

絶対値の小さな固有値から求めるには，行列式の逆行列 A^{-1} に累乗法を当てはめれば，その固有値の逆数が求める固有値である．

プログラム (Eigen_Power.java) のメソッド (eigen_Power) と，その説明は以下のとおりである．

Eigen_Power.java のメソッド (eigen_Power)：

```
 1  public void eigen_Power(double[][] a, double[][] s, double[] d,
          int itr) {
 2    int n= a.length;
 3    int m= d.length;
 4    double dmax;
 5    double[] s0= new double[n];
 6    s0[0]= 1.0;
 7    for (int i=1; i<n; i++) {
 8      s0[i]= 0.0;
 9    }
10    for (int k=0; k<m; k++) {
11      dmax= eigen_Max_Power(a, s0, itr);
12      d[k]= dmax;
13      for (int i=0; i<n; i++) {
14        s[i][k]= s0[i];
15      }
16      a= mat_Sub(a, c_Mul(dmax, direct_Product(s0, s0)));
17    }
18  }
```

1. L1〜18: メソッド eigen_Power の定義である．固有値問題の式は $AS = SD$ であり，引数の a は行列 A，s は固有ベクトルが縦ベクトルとなっている行列 S，d は固有値のベクトル $\mathbf{d}$ である．itr は収束しない場合の反復回数の上限である．
2. L2: n は行列の次元数.
3. L3: m は求める固有値の数であり，大きいほうの m 個の固有値が求められる.
4. L4: 求める最大固有値を代入する変数.
5. L5〜9: s0 は最大固有値の固有ベクトルの配列であり，初期化しておく.
6. L10: 求める固有値の数 m だけループをまわす.
7. L11: 最大固有値を求めるメソッド eigen_Max_Power を呼ぶ．引数は行列 a，固有ベクトル s0，反復回数の上限 itr である．戻り値の固有値が dmax に代入される.
8. L12〜15: 固有値と固有ベクトルを，それぞれ配列 d と s に代入する.
9. L16: 式 (9.36) の処理を行い，次の最大固有値を求めるための行列 a を作る．このクラスには，行列の引き算 mat_Sub，数を行列に掛ける c_Mul とベクトルの直積 direct_Product がメソッドとして用意されている.
10. L18: 計算結果の固有値と固有ベクトルは引数の配列 d と s で戻される.

プログラム (Eigen_Power.java) のメソッド (eigen_Max_Power) と，その説明は以下のとおりである.

Eigen_Power.java のメソッド (eigen_Max_Power)：

```
 1  public double eigen_Max_Power(double[][] a, double[] s0,
          int itr) {
 2    int n= a.length;
 3    double[] sw= new double[n];
 4    int m= 0;
```

```
 5     double dmax= 0.0;
 6     double d;
 7     double snorm= 0.0;
 8     snorm= dot_Product(s0, s0);
 9     snorm= Math.sqrt(snorm);
10     for (int i=0; i<n; i++) {
11       s0[i] /= snorm;
12     }
13     do {
14       m++;
15       sw= mat_Mul(a, s0);
16       double swsw= dot_Product(sw, sw);
17       double s0sw= dot_Product(s0, sw);
18       d= swsw/s0sw;
19       if (Math.abs(dmax-d)<1.0e-6) break;
20       dmax= d;
21       double swnorm= Math.sqrt(swsw);
22       for (int i=0; i<n; i++) {
23         s0[i]= sw[i]/swnorm;
24       }
25     } while (m<itr);
26     if (m>=itr) System.out.println("ERROR No Convergence, m= "+m);
27     return dmax;
28   }
```

1. L1〜28: メソッド eigen_Max_Power の定義である．固有値問題の式は $AS = SD$ であり，引数の a は行列 A，s0 は求める最大固有値のための固有ベクトルである．固有値は戻り値で返される．引数の itr は収束しない場合の反復回数の上限である．
2. L2: n は行列の次元数である．
3. L3: sw は反復するときの新しい固有ベクトルの配列である．
4. L4: m は反復回数のカウンタでゼロに初期化する．
5. L7〜12: ベクトル s0 のノルムを 1 に規格化する．
6. L8: このクラスにあるメソッド dot_Product であり，引数の 2 個のベクトルの内積を返す．
7. L13〜25: 求める固有値が収束するまで A の累乗 (L15) のループをまわす．
8. L14: カウンタに 1 を加える．
9. L15〜18: 式 (9.35) の d を求める．
10. L15: このクラスにあるメソッド mat_Mul であり，引数の行列とベクトルの積のベクトルを返す．
11. L19: 収束していたらループを終了する．
12. L20〜24: まだ，収束していなかったら，dmax と新しい s0 を準備する．
13. L25, 26: 反復回数が上限に達したらループを終了し，メッセージを出力する．
14. L27: 固有値を戻り値として返す．

9.2.5 QR 法

ヤコビ法は単純明快であるが，処理効率はあまり良いとはいえず，$n > 10$ になると実用的でなくなり，これまでに述べた他の方法とともに古典的方法といわれることもある．ヤコビ法で非対角要素をゼロにしていく過程は，値の大きい要素からゼロにしていくが，一度ゼロにした要素も他の要素をゼロにする過程で再び有限の値をもつようになり，あたかもモグラ叩きをする感がある．

現在 n が大きい固有値問題では，初めの行列の非対角要素にゼロが少ない密行列の場合や，ゼロが多い疎行列の場合などで，それぞれ有利な方法が考えられている．もちろん，ベクトルプロセッサを使わければならないが，5 万次元で非零行列要素数が 5000 万の行列の対角化が，ランチョス法などによって行われている．

これから述べる方法は，密行列に使える方法であり，まず，ハウスホールダー法を使って，対称行列の場合には 3 重対角行列に，非対称行列の場合には上ハッセンベルグ行列に変換し，その後，QR 法によって対角行列，あるいは上 3 角行列を求める方法である．ハウスホールダー法の過程は，モグラ叩きとは異なり，各列について一通り処理すれば済み，QR 法の過程では繰り返しが行われるが，ヤコビ法などに比べれば，非常に効率が良いといえる．

次のプログラムは，固有値を求めるだけなので，固有ベクトルを必要としたり，大きな問題を扱う場合には，効率が良くなるように研究されているパッケージのソフトウェアを利用して頂きたい [20][21].

ハウスホールダー法

最初の n 次元行列

$$A_0 = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ a_{31} & a_{32} & \cdots & a_{3n} \\ \cdot & \cdot & \cdots & \cdot \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \tag{9.39}$$

にハウスホールダー法を適用して，上ハッセンベルグ行列にする．そのために，まず，A の第 1 列の第 3 行目から下がゼロの行列

$$A_1 = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ 0 & a_{32} & \cdots & a_{3n} \\ \cdot & \cdot & \cdots & \cdot \\ 0 & a_{n2} & \cdots & a_{nn} \end{pmatrix} \tag{9.40}$$

に変換する直交変換のことを考える．そのような直交変換は対称な直交行列 S_1 を使うと，

$$
\begin{aligned}
A_1 &= S_1^{-1} A_0 S_1 \\
&= \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & s_{22} & \cdots & s_{2n} \\ \cdot & \cdot & \cdots & \cdot \\ 0 & s_{n2} & \cdots & s_{nn} \end{pmatrix} \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ a_{21} & \cdots & a_{2n} \\ \cdot & \cdots & \cdot \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & s_{22} & \cdots & s_{2n} \\ \cdot & \cdot & \cdots & \cdot \\ 0 & s_{n2} & \cdots & s_{nn} \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ 0 & S_1^{(22)} \end{pmatrix} \begin{pmatrix} a_{11} & \mathbf{x}_1^{(12)T} \\ \mathbf{x}_1^{(21)} & A_0^{(22)} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & S_1^{(22)} \end{pmatrix} \\
&= \begin{pmatrix} a_{11} & (S_1^{(22)} \mathbf{x}_1^{(12)})^T \\ S_1^{(22)} \mathbf{x}_1^{(21)} & S_1^{(22)} A_0^{(22)} S_1^{(22)} \end{pmatrix}
\end{aligned}
\tag{9.41}
$$

となるから,

$$
\mathbf{y}_1^{(21)} = S_1^{(22)} \mathbf{x}_1^{(21)} = \begin{pmatrix} \sigma_1 \\ 0 \\ \cdot \\ 0 \end{pmatrix} \tag{9.42}
$$

となるように S_1 を作ればよい．ただし，$A_0^{(22)}$ は行列 A_0 の要素 a_{22} を含めた右下の小行列であり，$S_1^{(22)}$ も同様である．また,

$$
\mathbf{x}_1^{(21)} = \begin{pmatrix} a_{21} \\ \cdot \\ \cdot \\ a_{n1} \end{pmatrix}, \qquad \mathbf{x}_1^{(12)} = \begin{pmatrix} a_{12} \\ \cdot \\ \cdot \\ a_{1n} \end{pmatrix} \tag{9.43}
$$

である．

式 (9.42) を満たす S_1 は次のように作ればよい．簡単化のために式 (9.42) を

$$
\mathbf{y} = S\mathbf{x} \tag{9.44}
$$

のように書き換える．ベクトル $\mathbf{x}$ と $\mathbf{y}$ の長さは等しくしておく．ここで,

$$
S = I - \frac{2}{|\mathbf{x}-\mathbf{y}|^2} (\mathbf{x}-\mathbf{y})(\mathbf{x}-\mathbf{y})^T \tag{9.45}
$$

と置いてみる．すると,

$$
\begin{aligned}
S\mathbf{x} - \mathbf{y} &= \left(I - \frac{2}{|\mathbf{x}-\mathbf{y}|^2} (\mathbf{x}-\mathbf{y})(\mathbf{x}-\mathbf{y})^T \right) \mathbf{x} - \mathbf{y} \\
&= (\mathbf{x}-\mathbf{y}) - \frac{2}{|\mathbf{x}-\mathbf{y}|^2} (\mathbf{x}-\mathbf{y})(\mathbf{x}^T - \mathbf{y}^T)\mathbf{x} \\
&= -\frac{1}{|\mathbf{x}-\mathbf{y}|^2} (\mathbf{x}-\mathbf{y}) \left(-(\mathbf{x}^T - \mathbf{y}^T)(\mathbf{x}-\mathbf{y}) + 2(\mathbf{x}^T - \mathbf{y}^T)\mathbf{x} \right) \\
&= -\frac{1}{|\mathbf{x}-\mathbf{y}|^2} (\mathbf{x}-\mathbf{y})(\mathbf{x}^T\mathbf{x} + \mathbf{x}^T\mathbf{y} - \mathbf{y}^T\mathbf{x} - \mathbf{y}^T\mathbf{y}) \\
&= \mathbf{0}
\end{aligned}
\tag{9.46}
$$

となり，確かに式 (9.44) が成り立っている．式 (9.45) の転置をとると S に戻るので $S^T = S$ であり，また，

$$\begin{aligned} SS^T &= \left(I - \frac{2}{|\mathbf{x}-\mathbf{y}|^2}(\mathbf{x}-\mathbf{y})(\mathbf{x}-\mathbf{y})^T\right)\left(I - \frac{2}{|\mathbf{x}-\mathbf{y}|^2}(\mathbf{x}-\mathbf{y})(\mathbf{x}-\mathbf{y})^T\right)^T \\ &= I - \frac{4}{|\mathbf{x}-\mathbf{y}|^2}(\mathbf{x}-\mathbf{y})(\mathbf{x}-\mathbf{y})^T \\ &\quad + \frac{4}{|\mathbf{x}-\mathbf{y}|^4}(\mathbf{x}-\mathbf{y})(\mathbf{x}-\mathbf{y})^T(\mathbf{x}-\mathbf{y})(\mathbf{x}-\mathbf{y})^T \\ &= I = SS^{-1} \end{aligned} \tag{9.47}$$

であるから，S は対称な直交行列である．以上のことを式 (9.42) に適用すると，

$$\mathbf{d}_1^{(21)} = \mathbf{x}_1^{(21)} - \mathbf{y}_1^{(21)} = \begin{pmatrix} a_{21} - \sigma_1 \\ \cdot \\ \cdot \\ a_{n1} \end{pmatrix} \tag{9.48}$$

であるから，

$$S_1^{(22)} = I - \frac{2}{|\mathbf{d}_1|^2}\mathbf{d}_1\mathbf{d}_1^T \tag{9.49}$$

となる．σ_1 は $\mathbf{x}_1^{(21)}$ と $\mathbf{y}_1^{(21)}$ の長さが等しいという条件から，$\sigma_1 = \pm\sqrt{\sum_{i=2}^{n} a_{i1}}$ であるが，符号は $|\mathbf{x}-\mathbf{y}|$ が小さくなって桁落ち誤差が生じないように，a_{21} と異符号のものを使う．

これで，A の第 1 列の第 3 行目から下がゼロとなった行列 A_1 が求められたが，続いて第 2 列の第 4 行目から下をゼロとする行列 S_2 で直交変換を行い，S_{n-2} まで繰り返せばよい．そこで，

$$S = \prod_{i=1}^{n-2} S_i \tag{9.50}$$

と置けば，上ハッセンベルグ行列は

$$A_{n-2} = S_{-1}AS \tag{9.51}$$

である．

式 (9.45) において，

$$\mathbf{d} = (\mathbf{x}-\mathbf{y})/|\mathbf{x}-\mathbf{y}| \tag{9.52}$$

とし，1 回の直交変換を記すと，

$$\begin{aligned} SAS &= (I - 2\mathbf{d}\mathbf{d}^T)A(I - 2\mathbf{d}\mathbf{d}^T) \\ &= A - 2\mathbf{d}\mathbf{d}^T A - 2A\mathbf{d}\mathbf{d}^T + 4\mathbf{d}\mathbf{d}^T A\mathbf{d}\mathbf{d}^T \\ &= A - 2(A\mathbf{d} - \mathbf{d}\mathbf{d}^T A\mathbf{d})\mathbf{d}^T - 2\mathbf{d}(\mathbf{d}^T A - \mathbf{d}^T A\mathbf{d}\mathbf{d}^T) \end{aligned} \tag{9.53}$$

である．プログラムの際には，できるだけ「行列掛ける行列」で括らないようにする必要がある．

QR 法

上ハッセンベルグ行列 A ができたので，次は，QR 法によって上三角行列にする．変換を行う直交行列として，ギブンスの直交行列

$$Q_i = \begin{pmatrix} 1 & \cdot & \cdot & \cdot & \cdots & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cos\theta_i & \sin\theta_i & \cdots & 0 \\ \cdot & \cdot & -\sin\theta_i & \cos\theta_i & \cdots & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdots & \cdot \\ 0 & \cdot & 0 & 0 & \cdots & 1 \end{pmatrix} \tag{9.54}$$

を使う．ただし，Q_i は $i = 1 \sim (n-1)$ である．ヤコビ法では，直交変換後の行列 S^TAS の該当する非対角要素をゼロにしたが，それではすでにゼロになっている他の非対角要素の値が復活するので，収束が遅かった．ここでは，まず $Q_1A = R_1$ として $r_{21} = 0$ となるように θ_1 を決める．この操作は，

$$\begin{aligned} R_1 &= Q_1A \\ &= \begin{pmatrix} \cos\theta_1 & \sin\theta_1 & \cdots & 0 & 0 \\ -\sin\theta_1 & \cos\theta_1 & \cdots & 0 & 0 \\ \cdot & \cdot & \cdots & \cdot & \cdot \\ 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n-1} & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n-1} & a_{2n} \\ 0 & a_{32} & \cdots & a_{3n-1} & a_{3n} \\ \cdot & \cdot & \cdots & \cdot & \cdot \\ 0 & 0 & \cdots & a_{nn-1} & a_{nn} \end{pmatrix} \\ &= \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n-1}^{(1)} & a_{1n}^{(1)} \\ 0 & a_{22}^{(1)} & \cdots & a_{2n-1}^{(1)} & a_{2n}^{(1)} \\ 0 & a_{32} & \cdots & a_{3n-1} & a_{3n} \\ \cdot & \cdot & \cdots & \cdot & \cdot \\ 0 & 0 & \cdots & a_{nn-1} & a_{nn} \end{pmatrix} \end{aligned} \tag{9.55}$$

である．変換される要素を具体的に書くと，

$$\begin{aligned} a_{11}^{(1)} &= a_{11}\cos\theta_1 + a_{21}\sin\theta_1 \\ a_{21}^{(1)} &= -a_{11}\sin\theta_1 + a_{21}\cos\theta_1 = 0 \\ a_{1j}^{(1)} &= a_{1j}\cos\theta_1 + a_{2j}\sin\theta_1 \quad (j = 2, \cdots, n-1) \\ a_{2j}^{(1)} &= -a_{1j}\sin\theta_1 + a_{2j}\cos\theta_1 \quad (j = 2, \cdots, n-1) \end{aligned} \tag{9.56}$$

であり，第 1,2 行のみが変更される．この式の最初の 2 式と θ_1 に対するピタゴラスの定理から，

$$\begin{aligned} a_{11}^{(1)} &= \sqrt{a_{11}^2 + a_{21}^2} \\ \cos\theta_1 &= a_{11}/a_{11}^{(1)} \\ \sin\theta_1 &= a_{21}/a_{11}^{(1)} \end{aligned} \tag{9.57}$$

が定まる．同様なことを Q_{n-1} まで行うと，

$$R = Q_{n-1} \cdots Q_2 Q_1 A = \begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1n-1} & r_{1n} \\ 0 & r_{22} & \cdots & r_{2n-1} & r_{2n} \\ 0 & 0 & \cdots & r_{3n-1} & r_{3n} \\ \cdot & \cdot & \cdots & \cdot & \cdot \\ 0 & 0 & \cdots & 0 & r_{nn} \end{pmatrix} \tag{9.58}$$

は上三角行列になる．ここで，後の便宜のために $Q = (Q_{n-1} \cdots Q_2 Q_1)^{-1}$ と置くと

$$R = Q^{-1} A \rightarrow A = QR \tag{9.59}$$

となる．このように A を直交行列 Q と上三角行列 R に分解することを QR 分解という．

次に A の Q による直交変換を作ると

$$A' = Q^{-1} A Q = RQ = \begin{pmatrix} a'_{11} & a'_{12} & \cdots & a'_{1n-1} & a'_{1n} \\ r_{22} \sin\theta_1 & a'_{22} & \cdots & a'_{2n-1} & a'_{2n} \\ 0 & r_{33} \sin\theta_2 & \cdots & a'_{3n-1} & a'_{3n} \\ \cdot & \cdot & \cdots & \cdot & \cdot \\ 0 & 0 & \cdots & r_{nn} \sin\theta_{n-1} & a'_{nn} \end{pmatrix} \tag{9.60}$$

となって上ハッセンベルグ行列が復活する．しかし，復活した各要素とも $|\sin\theta| \leq 1$ が掛かっているから，処理のたびに小さくなっていくであろうことがわかる．

もし，行列 A の要素 $a_{n(n-1)}$ がゼロに (閾値より小さく) なったら，次は，行列 A の左上の $n-1$ 次の小行列での処理をすればよく，順次，次数が減って最後に左上の 2 次の小行列を処理して終了する．

プログラム (Eigen_QR.java) のメソッド (house) と，その解説は以下のとおりである．

Eigen_QR.java のメソッド (house)：

```
 1  void house(double[][] a, double eps) {
 2    int n= a.length;
 3    double w, sigma, norm, sum, dtad;
 4    double[] x= new double[n];
 5    double[] ad= new double[n];
 6    double[] dta= new double[n];
 7    int ip= 0;
 8
 9    for (int k=0; k<n-2; k++) {
10      w= 0.0;
11      for (int i=k+1; i<n; i++) {  //  k 列の k+1 行以下で最大を探す.
12        if (Math.abs(a[i][k])  >  w) {
13          w = Math.abs(a[i][k]);
14          ip = i;
15        }
```

```
16        }
17        if (ip != k+1) {  //  ピボットが k+1 行でなかったら入れ替える
18          for (int j=0; j<n; j++) {
19            w= a[k+1][j];
20            a[k+1][j]= a[ip][j];
21            a[ip][j]= w;
22          }
23          for (int j=0; j<n; j++) {
24            w= a[j][k+1];
25            a[j][k+1]= a[j][ip];
26            a[j][ip]= w;
27          }
28        }
29
30        for (int i=0; i<=k; i++) {
31          x[i]= 0.0;
32        }
33
34        for (int i=k+1; i<n; i++) {
35          x[i]= a[i][k];
36        }
37        if (Math.abs(x[k+1])<eps) continue;
38        sum= 0.0;
39        for (int i=k+1; i<n; i++) {
40          sum += x[i]*x[i];
41        }
42        sigma= -Math.sqrt(sum)*x[k+1]/Math.abs(x[k+1]);
43        x[k+1] -= sigma;
44        norm= Math.sqrt(-2*sigma*x[k+1]);
45        for (int i=k+1; i<n; i++) {
46          x[i] /= norm;
47        }
48
49        for (int i=0; i<n; i++) {
50          ad[i]= dta[i]= 0.0;
51          for (int j=k+1; j<n; j++) {
52            ad[i] += a[i][j]*x[j];
53            dta[i] += x[j]*a[j][i];
54          }
55        }
56        dtad= 0.0;
57        for (int i=k+1; i<n; i++) {
58          dtad += ad[i]*x[i];
59        }
60        for (int i=0; i<n; i++) {
61          ad[i]= 2*(ad[i] - x[i]*dtad);
62          dta[i]= 2*(dta[i] - dtad*x[i]);
63        }
64        for (int i=0; i<n; i++) {
65          for (int j=0; j<n; j++) {
66            a[i][j] -= (ad[i]*x[j]+x[i]*dta[j]);
```

```
67          }
68        }
69      }
70    }
```

1. L1〜70: プログラム Eigen_QR.java のメソッド house の定義である．引数の a は上ハッセンベルグ行列化する元の行列であり，eps はゼロにする要素の値の閾値である．
2. L4: x は最初は式 (9.44) の $\mathbf{x}$ であるが，L35 で式 (9.48) の $\mathbf{d}$ の一般型を規格化したものとなる．
3. L5, 6: $A\mathbf{d}$，$\mathbf{d}A$ など途中の作業に使う配列である．
4. L9〜70: 列 k に関するループである．
5. L10〜28: ピボット選択をする．
6. L30〜36: 式 (9.43) の $\mathbf{x}_1^{(21)}$ を作る．
7. L37: すでに小さかったらループを先に進める．
8. L38〜47: 式 (9.48) の $\mathbf{d}$ の一般型を規格化したものを作る．
9. L49〜68: 式 (9.53) の直交変換を行う．

プログラム (Eigen_QR.java) のメソッド (eigen_QR) と，その解説は以下のとおりである．

Eigen_QR.java のメソッド (eigen_QR)：

```
 1 public double[] eigen_QR(double[][] a, double eps) {
 2     int n= a.length;
 3     double[] d= new double[n];
 4     double[][] q= new double[n][n];
 5     double[] w= new double[n];
 6     double sum1, sum2, wa;
 7     double sint, cost;
 8     house(a, eps);
 9     int m= n;
10     while (m!=1) {
11       if (Math.abs(a[m-1][m-2])<eps) {
12         m= m-1;
13        continue;
14       }
15       for (int i=0; i<n; i++) {
16         for (int j=0; j<n; j++) {
17           q[i][j]= 0.0;
18         }
19         q[i][i]= 1.0;
20       }
21       for (int i=0; i<m-1; i++) {
22         sum1= Math.sqrt(a[i][i]*a[i][i]+a[i+1][i]*a[i+1][i]);
23         if (Math.abs(sum1)<eps) {
24           sint= 0.0;
25           cost= 0.0;
26         } else {
```

```
27          sint= a[i+1][i]/sum1;
28          cost= a[i][i]/sum1;
29        }
30        for (int j=i+1; j<m; j++) {
31          sum2= a[i][j]*cost+a[i+1][j]*sint;
32          a[i+1][j]= -a[i][j]*sint+a[i+1][j]*cost;
33          a[i][j]= sum2;
34        }
35        a[i+1][i]= 0.0;
36        a[i][i]= sum1;
37        for (int j=0; j<m; j++) {
38          sum2= q[j][i]*cost+q[j][i+1]*sint;
39          q[j][i+1]= -q[j][i]*sint+q[j][i+1]*cost;
40          q[j][i]= sum2;
41        }
42      }
43
44      for (int i=0; i<m; i++) {
45        for (int k=i; k<m; k++) {
46          w[k]= a[i][k];
47        }
48        for (int j=0; j<m; j++) {
49          sum1= 0.0;
50          for (int k=i; k<m; k++) {
51            sum1 += w[k]*q[k][j];
52          }
53          a[i][j]= sum1;
54        }
55      }
56      for (int i=0; i<m; i++) {
57        d[i]= a[i][i];
58      }
59    }
60    return d;
61  }
```

1. L1〜61: プログラム Eigen_QR.java のメソッド eigen_QR の定義である．引数の a は固有値を求める行列であり，eps はゼロにする要素の値の閾値である．戻り値は固有値である．
2. L3: 戻り値の固有値の配列である．
3. L8: メソッド house を呼んで，a を上ハッセンベルグ行列にする．
4. L9: 最後の a[n−1][n−2] から処理を始めるために m に n を代入する．
5. L11〜14: すでに小さかったら m を 1 減らす．
6. L15〜20: n 次元の単位行列 q を作る．
7. L21〜42: 式 (9.55) の処理を 0 列から m−1 列まで繰り返す．
8. L22〜29: 式 (9.57) の計算をする．
9. L30〜36: 式 (9.56) の計算をする．
10. L37〜41: Q を作る．
11. L44〜55: 式 (9.60) の計算をする．

演習問題

9.1　次の行列の固有値と固有ベクトルをヤコビ法を使う JacobiTest.java を使って求めなさい.

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 2 & 3 & 4 \\ 3 & 3 & 3 & 4 \\ 4 & 4 & 4 & 4 \end{pmatrix}$$

答：

固有値	固有ベクトル			
−0.29432	0.276677	−0.66335	0.650424	−0.24566
12.86208	0.410342	0.442245	0.508532	0.614356
−0.51464	−0.51437	0.485103	0.541978	−0.45426
−2.05311	−0.70034	−0.35923	0.156851	0.596539

第 10 章

フーリエ級数とフーリエ変換

ニュートンの補間公式やラグランジュの補間公式，あるいはチェビシェフ近似は，べき級数展開の低次の数項を使うものであったが，周期的な性質のあるデータの場合には，三角関数で級数展開するフーリエ級数が使われる．この場合は，関数を少数の低次の項だけでは良く近似することが不可能なことが多く，近似のために使われるということはあまりないが，与えられたシグナルなどの関数に，どのような周期の波がどの程度混ざっているかを調べるスペクトル解析などによく使われる．この場合は，フーリエ解析ともいわれる．また，量子力学の分野では，電子の波動関数のフーリエ変換である運動量波動関数などが，実空間での関数同様に重要な概念となっている．一般的には，与えられるデータは解析的な関数 $f(x)$ ではなく，離散的な数値データ $f(x_i)$ である場合が多く，その場合には変数 x の範囲は有限であり，フーリエ変換の積分領域も有限となる．この場合のフーリエ変換は，離散フーリエ変換 (DFT) といわれる．

10.1　フーリエ級数

まず，簡単な例として，周期関数である鋸波と矩形波の場合を取り上げる．図 10.1 は周期関数である鋸波 (実線) のフーリエ級数展開の収束の様子を示している．鋸波とそのフーリエ展開は

$$f(x) = \left\{ \begin{array}{cc} & \cdots \\ \dfrac{(x+2\pi)}{\pi} & (-3\pi < x \le -\pi) \\ \dfrac{x}{\pi} & (-\pi < x \le \pi) \\ \dfrac{(x-2\pi)}{\pi} & (\pi < x \le 3\pi) \\ & \cdots \end{array} \right\} = \frac{2}{\pi}\sum_{k=1}^{\infty}(-1)^{k-1}\frac{1}{k}\sin kx \tag{10.1}$$
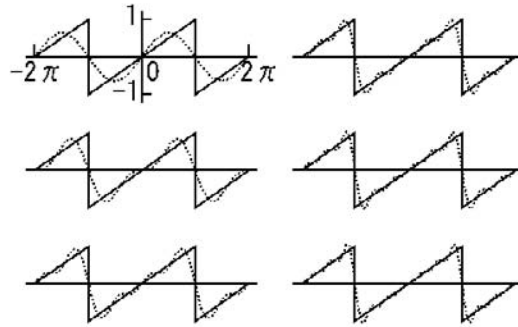
図 10.1　鋸波 (実線) のフーリエ級数展開．左上は第 1 項 $\frac{2}{\pi}\sin x$ のみであり，左中，左下，右上などの順に項の数が 1 項ずつ増加している．右下は第 6 項までである．

である．図の左上は第 1 項 $\frac{2}{\pi}\sin x$ のみであり，左中，左下，右上などの順に項の数が 1 項ずつ増加している．

この場合は，元の関数 (鋸波) が奇関数であるから，級数は sin 関数のみであり，cos 関数は含まれていない．

この場合は，元の関数 (矩形波) が偶関数であるから，級数は cos 関数のみであり，sin 関数は含まれていない．その上，元の関数は 1 周期が 2π であり，値が $+1$ と -1 である領域が π ずつで等しいから，1 周期の中に波数が偶数個ある $\cos 2kx$ の項のフーリエ係数はゼロとなり，それらの項は現れてこない．

図 10.2 は周期関数である矩形波 (実線) のフーリエ級数展開の収束の様子を示している．矩形波とそのフーリエ展開は
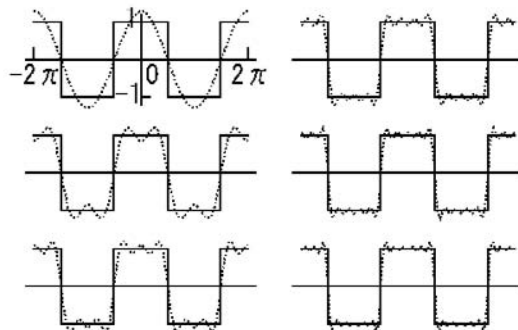
図 10.2　矩形波 (実線) のフーリエ級数展開．左上は第 1 項 $\frac{4}{\pi}\cos x$ のみであり，左中，左下，右上などの順に項の数が 1 項ずつ増加している．右下は第 6 項までである．

$$
\begin{aligned}
f(x) &= \left\{\begin{array}{ll}
-1 & (-\pi < x < -\frac{\pi}{2},\ \frac{\pi}{2} < x < \pi) \\
0 & (x = -\frac{\pi}{2},\ \frac{\pi}{2}) \\
1 & (-\frac{\pi}{2} < x < \frac{\pi}{2})
\end{array}\right\} \\
&= \frac{4}{\pi}\sum_{k=1}^{\infty}(-1)^{k-1}\frac{1}{2k-1}\cos(2k-1)x
\end{aligned}
\tag{10.2}
$$

である．図の左上は第 1 項 $\frac{4}{\pi}\cos x$ のみであり，左中，左下，右上などの順に項の数が 1 項ずつ増加している．

10.1.1　三角関数による級数展開

周期が 2π の関数 $f(x)$ の値が等間隔の n 点 $x_j = \frac{2\pi j}{n}$ $(j = 0, 1, \cdots, n-1)$ について与えられているとき，この n 点を通る補間式は，低次の n 個の三角関数を使って

$$
f(x) = \left\{\begin{array}{l}
\frac{a_0}{2} + \sum_{k=1}^{m-1}(a_k \cos kx + b_k \sin kx) + \frac{a_m}{2}\cos mx \\
\qquad (m = \frac{n}{2},\ n \text{ が偶数の場合}) \\
\frac{a_0}{2} + \sum_{k=1}^{m-1}(a_k \cos kx + b_k \sin kx) \\
\qquad (m = \frac{n+1}{2},\ n \text{ が奇数の場合})
\end{array}\right.
\tag{10.3}
$$

とし，フーリエ係数 (a_k, b_k) を決定すればよい．n が偶数の場合には，次の積和

$$
\begin{aligned}
\sum_{j=0}^{n-1} f(x_j) &= \sum_{j=0}^{n-1}\left\{\frac{a_0}{2} + \sum_{k=1}^{m-1}(a_k\cos kx_j + b_k\sin kx_j) + \frac{a_m}{2}\cos mx_j\right\} \\
&= \frac{n}{2}a_0
\end{aligned}
\tag{10.4}
$$

$$
\begin{aligned}
&\sum_{j=0}^{n-1}\cos kx_j f(x_j) \\
&\quad= \sum_{j=0}^{n-1}\cos kx_j\left\{\frac{a_0}{2} + \sum_{k=1}^{m-1}(a_k\cos kx_j + b_k\sin kx_j) + \frac{a_m}{2}\cos mx_j\right\} \\
&\quad= \frac{n}{2}a_k
\end{aligned}
\tag{10.5}
$$

$$
\begin{aligned}
&\sum_{j=0}^{n-1}\sin kx_j f(x_j) \\
&\quad= \sum_{j=0}^{n-1}\sin kx_j\left\{\frac{a_0}{2} + \sum_{k=1}^{m-1}(a_k\cos kx_j + b_k\sin kx_j) + \frac{a_m}{2}\cos mx_j\right\} \\
&\quad= \frac{n}{2}b_k
\end{aligned}
\tag{10.6}
$$

から，

$$a_0 = \frac{2}{n}\sum_{j=0}^{n-1} f(x_j), \quad a_k = \frac{2}{n}\sum_{j=0}^{n-1}\cos kx_j f(x_j), \quad b_k = \frac{2}{n}\sum_{j=0}^{n-1}\sin kx_j f(x_j) \tag{10.7}$$

が決まる．n が奇数の場合も同様である．式 (10.3) の両式とも項の数は n であり，使用するデータ点の数に等しい．使われている項以外には独立な項がないことは次のことから理解できる．まず，$0 \le k' < n$ とすると，任意の整数 i についての $k = k' + in$ に関して

$$\begin{aligned}
\cos kx_j &= \cos(k' + in)x_j = \cos k'x_j \cos inx_j - \sin k'x_j \sin inx_j = \cos k'x_j \\
\sin kx_j &= \sin(k' + in)x_j = \sin k'x_j \cos inx_j + \cos k'x_j \sin inx_j = \sin k'x_j
\end{aligned} \tag{10.8}$$

であるから，$0 \le k' < n$ の範囲で調べればよい．式 (10.3) の第 1 式 (n が偶数) において，$m \le k < n$ の場合には，$k = n - k'$ と置くと $0 < k' \le m$ となり

$$\begin{aligned}
\cos kx_j &= \cos(n - k')x_j = \cos nx_j \cos k'x_j + \sin nx_j \sin k'x_j \\
&= \cos 2\pi j \cos k'x_j + \sin 2\pi j \sin k'x_j = \cos k'x_j \\
\sin kx_j &= \sin(n - k')x_j = \sin nx_j \cos k'x_j - \cos nx_j \sin k'x_j \\
&= \sin 2\pi j \cos k'x_j - \cos 2\pi j \sin k'x_j = -\sin k'x_j
\end{aligned} \tag{10.9}$$

であるから，両関数ともすでに使われている．特に，$k = m$ のときは

$$\sin mx_j = \sin \frac{n}{2}\frac{2\pi j}{n} = \sin \pi j = 0 \tag{10.10}$$

であるから，$\sin mx_j$ は使われていない．式 (10.3) の第 2 式 (n が奇数) においても同様である．

式 (10.3) の関数 $f(x)$ は複素関数でもよいが，その場合には係数も複素数となる．そこで，三角関数の代わりに指数関数 e^{ix} を使うことが多い．

10.1.2　指数関数 e^{ix} による展開

オイラーの公式は，

$$e^{ix} = \cos x + i \sin x \quad (i = \sqrt{-1} \quad \text{虚数単位}) \tag{10.11}$$

である．ここで，

$$\begin{aligned}
c_0 &= \frac{a_0}{2} \\
c_k &= \frac{a_k - ib_k}{2} && (k = 1, \cdots, m-1) \\
c_m &= \frac{a_m}{2} && (n\ \text{が偶数の場合}) \\
c_k &= \frac{a_{n-k} + ib_{n-k}}{2} && (k = m+1, \cdots, n-1)
\end{aligned} \tag{10.12}$$

とし，さらに，

$$f_h = f\left(\frac{2\pi h}{n}\right) \quad (h = 0, 1, \cdots, n-1)$$
$$\omega = e^{-\frac{2\pi}{n}i} = \cos\left(-\frac{2\pi}{n}\right) + i\sin\left(-\frac{2\pi}{n}\right) = \cos\left(\frac{2\pi}{n}\right) - i\sin\left(\frac{2\pi}{n}\right) \tag{10.13}$$

とすると，式 (10.11) を使って式 (10.3) は

$$f_h = \sum_{k=0}^{n-1} c_k \omega^{-kh} \tag{10.14}$$

となる．a_k, b_k を求める手続きと同じように，この式の両辺に $\omega^{k'h}$ を掛けて h について和をとると，

$$\sum_{h=0}^{n-1} \omega^{k'h} f_h = \sum_{h=0}^{n-1}\sum_{k=0}^{n-1} c_k \omega^{k'h}\omega^{-kh} = nc_{k'} \tag{10.15}$$

すなわち，k' を k として

$$c_k = \frac{1}{n}\sum_{h=0}^{n-1} \omega^{kh} f_h \quad (k = 0, \cdots, n-1) \tag{10.16}$$

となる．この際，

$$\sum_{h=0}^{n-1} \omega^{(k-k')h} = \begin{cases} n & (k = k') \\ 0 & (k \neq k') \end{cases} \tag{10.17}$$

を使っている．

10.2　フーリエ変換

関数 $f(x)$ が $[-\infty \leq x \leq \infty]$ で定義されているとき，

$$g(q) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x) e^{-qx} \mathrm{d}x \tag{10.18}$$

を $f(x)$ のフーリエ変換という．すると，逆に

$$f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(q) e^{qx} \mathrm{d}q \tag{10.19}$$

の関係もあり，$g(q)$ は $f(x)$ の逆フーリエ変換という．フーリエ変換と逆フーリエ変換の相違は被積分関数の中の指数関数に $-i$ か i のどちらが使われているかの相違だけであるから，逆の定義でも全体としては矛盾が生じない．また，積分の前の係数 $\dfrac{1}{\sqrt{2\pi}}$ の付け方も定義次第であり，片方に $\dfrac{1}{2\pi}$ を付け他方には何も付けないこともある．ここでは式 (10.18) と式 (10.19) を採用する．

プログラム (Fourier_Transform.java) のメソッド (fourier_Transform) と，その説明は以下のとおりである．

Fourier_Transform.java のメソッド (fourier_Transform)：

```
 1  public double fourier_Transform(double[] x, MyFunction f) {
 2    int nx= x.length;
 3    Gauss_Legendre gl= new Gauss_Legendre(16);
 4    double xmin, xmax, sum;
 5    double C= 1.0/Math.sqrt(2*Math.PI);
 6    sum= 0.0;
 7    xmin= x[0];
 8    for (int i=1; i<nx; i++) {
 9      xmax= x[i];
10      sum += gl.gauss_Legendre(xmin, xmax, f);
11      xmin= xmax;
12    }
13  return C*sum;
14  }
```

1. L1〜14: メソッド fourier_Transform の定義である．積分は積分領域を適当に分割し，Gauss_Legendre(16) 積分を繰り返し行うため，配列 x に座標の分割点を指定する．f はインタフェース MyFunction クラスのインスタンスであり，解を求める関数が定義されている．戻り値はフーリエ変換値である．
2. L5: 積分項に掛かるフーリエ変換の係数である．
3. L6: gauss_Legendre による積分値を足し込んでいく変数をゼロで初期化する．
4. L7: gauss_Legendre による積分の下限値を指定する．
5. L8〜12: 積分の上限値，下限値をずらしながら，積分を配列 x の要素数 −1 回繰り返す．
6. L13: 積分に係数を掛けて，戻り値として返す．

ガウス関数のフーリエ変換

ガウス関数 (正規分布関数)

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \tag{10.20}$$

のフーリエ変換は，やはりガウス関数

$$g(q) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} e^{-qx} \mathrm{d}x = \frac{1}{\sqrt{2\pi}} e^{-\frac{\sigma^2 q^2}{2}} \tag{10.21}$$

である．元のガウス関数の式 (10.20) の σ は標準偏差ともいわれ，$f(\sigma) = e^{-0.5} f(0) = 0.6060\cdots f(0)$ である．ガウス関数は規格化されており，

$$\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \mathrm{d}x = 1 \tag{10.22}$$

であり，原点での値は $f(0) = \dfrac{0.3989\cdots}{\sigma}$ である．ガウス関数は偶関数であるから，フーリエ変換は $\cos(qx)$ による余弦フーリエ変換のみが値をもち，$\sin(qx)$ による正弦フーリエ変換はゼロになる．その余弦フーリエ変換は，先の式 (10.21) と同じで

$$g(q) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \cos qx \, \mathrm{d}x = \frac{1}{\sqrt{2\pi}} e^{-\frac{\sigma^2 q^2}{2}} \tag{10.23}$$

である．この際，公式

$$\int_0^\infty e^{-a^2x^2}\cos bx\,\mathrm{d}x = \frac{\sqrt{\pi}}{2a}e^{-\frac{b^2}{4a^2}} \tag{10.24}$$

を使っている．図 10.3 の上段はガウス関数であり，下段はそのフーリエ変換である．実線は $\sigma = 1.0$ であり，ガウス関数とフーリエ変換は同じ形である．原点での値は，$0.3989\cdots$ である．点線は $\sigma = 0.5$ である．ガウス関数は 1 に規格化されているので，幅が半分になっている反面，原点での値は 2 倍になっている．そのフーリエ変換は幅が 2 倍になっているが，原点での値は変化しない．破線は $\sigma = 2.0$ である．幅が 2 倍になっており，原点での値は半分になっている．そのフーリエ変換は幅が半分になっているが，原点での値はやはり同一である．
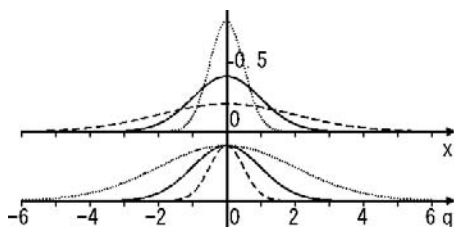
図 10.3　ガウス関数とフーリエ変換．上段はガウス関数 $\frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{x^2}{2\sigma^2}}$ であり，下段はそのフーリエ変換である．実線は $\sigma = 1.0$，点線は $\sigma = 0.5$，破線は $\sigma = 2.0$ である．

10.3　高速フーリエ変換（FFT）

式 (10.16) で f_h からフーリエ係数 c_k を求める変換を，離散フーリエ変換 (DFT) という．フーリエ係数を求めるために，式 (10.7) あるいは式 (10.16) をそのまま使って 1 個ずつ計算すると，n 項の積和を n 個計算しなければならず，計算処理は n^2 に比例することになる．しかし，式 (10.7) あるいは式 (10.16) を見ると，三角関数あるいは指数関数 $\omega^{kh} = e^{-\frac{2\pi i}{n}kh}$ は周期関数であり，しかも kh は整数であるから式 (10.16) の各 k に関する項の中には，異なる値となる $e^{-\frac{2\pi i}{n}kh}f_h$ は n 個だけである．クーリーとテューキーにより，データ点が等間隔であり，n が 2^m で 2 のべき乗であれば，f_h と位相因子の積和を工夫することによって，計算処理が nm に比例するアルゴリズムが創出され，それは，高速フーリエ変換 (Fast Fourier Transform) といわれている．

図 10.4 はガウス関数 ($\sigma = 1.0$) と FFT で求めたフーリエ変換である．ガウス関数は $[-6.0,\ 6.0]$ を 256 等分して値を求めてあり，実線で示されている．フーリエ変換の q 座標の刻み Δq は 2π を x 軸上の領域の幅で割ったものであるから，$\Delta q = \dfrac{2\pi}{6.0-(-6.0)} = 0.523\cdots$ であり，$[-6.0,\ 6.0]$ の領域に 256 点のうちの 23 点
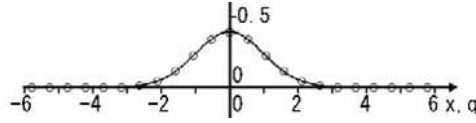
図 10.4 ガウス関数と FFT によるフーリエ変換．実線はガウス関数 ($\sigma = 1.0$) であり，$[-6.0,\ 6.0]$ が 256 等分されている．∘印はフーリエ変換であり，刻み Δq は $0.523\cdots$ である．(Z10_04_FFTC_gauss.java)

が∘印で示されている．q 座標での領域の幅は $\Delta q \times 256$ であるから，$[-134.0,\ 134.0]$ である．

表 10.1 は $n = 8$ の場合の式 (10.16) のフーリエ係数 $c_k = \dfrac{1}{n}\sum_{h=0}^{7}\omega^{kh} f_h$ の f_h に掛かる位相因子 ω^{kh} である．

$0 \leq h,\ k \leq 7$ であるから 2 進法の考えで，$k = p\,2^2 + q\,2 + r$ と $h = s\,2^2 + t\,2 + u$ と置くことが可能である．式 (10.16) の k, h をこの 2 進数的な数で置き換えて書きなおし，$\omega^8 = 1$ を使って整理すると

$$
\begin{aligned}
\sum_{h=0}^{7}\omega^{kh} f_h &= \sum_{s=0}^{1}\sum_{t=0}^{1}\sum_{u=0}^{1}\omega^{(p2^2+q2+r)(s2^2+t2+u)} f_{stu} \\
&= \sum_{s=0}^{1}\sum_{t=0}^{1}\sum_{u=0}^{1}\omega^{4rs}\omega^{(4q+2r)t}\omega^{(4p+2q+r)u} f_{stu} \qquad (10.25)\\
&= \sum_{u=0}^{1}\omega^{(4p+2q+r)u}\left\{\sum_{t=0}^{1}\omega^{(4q+2r)t}\left(\sum_{s=0}^{1}\omega^{4rs} f_{stu}\right)\right\}
\end{aligned}
$$

となる．

表 10.2〜表 10.5 は f_h から c_k への変換の過程を示すものである．表 10.2 は変換前の f_k であり，第 1 列は k の 2 進数 pqr で，まだ何も決まっていないから，– が入ってい

表 10.1 $n = 8$ の場合のフーリエ係数 c_k と関数値 f_h に掛かる位相因子 ω^{kh}．

c_k	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7
c_0	ω^0	ω^0	ω^0	ω^0	ω^0	ω^0	ω^0	ω^0
c_1	ω^0	ω^1	ω^2	ω^3	ω^4	ω^5	ω^6	ω^7
c_2	ω^0	ω^2	ω^4	ω^6	ω^8	ω^{10}	ω^{12}	ω^{14}
c_3	ω^0	ω^3	ω^6	ω^9	ω^{12}	ω^{15}	ω^{18}	ω^{21}
c_4	ω^0	ω^4	ω^8	ω^{12}	ω^{16}	ω^{20}	ω^{24}	ω^{28}
c_5	ω^0	ω^5	ω^{10}	ω^{15}	ω^{20}	ω^{25}	ω^{30}	ω^{35}
c_6	ω^0	ω^6	ω^{12}	ω^{18}	ω^{24}	ω^{30}	ω^{36}	ω^{42}
c_7	ω^0	ω^7	ω^{14}	ω^{21}	ω^{28}	ω^{35}	ω^{42}	ω^{49}

表 10.2　f_h と h の 2 進数 stu．第 1 列は k の 2 進数 pqr で，第 2 列は h の 2 進数 stu である．

$p\,q\,r$	$s\,t\,u$	
$---$	000	f_0
$---$	001	f_1
$---$	010	f_2
$---$	011	f_3
$---$	100	f_4
$---$	101	f_5
$---$	110	f_6
$---$	111	f_7

表 10.3　FFT の第 1 段階の変換．第 1 列は k の 2 進数 pqr で，第 2 列は h の 2 進数 stu である．

$p\,q\,r$	$s\,t\,u$	
$--0$	-00	f_0+f_4
$--1$	-00	$f_0+f_4\omega^4$
$--0$	-01	f_1+f_5
$--1$	-01	$f_1+f_5\omega^4$
$--0$	-10	f_2+f_6
$--1$	-10	$f_2+f_6\omega^4$
$--0$	-11	f_3+f_7
$--1$	-11	$f_3+f_7\omega^4$

表 10.4　FFT の第 2 段階の変換．第 1 列は k の 2 進数 pqr で，第 2 列は h の 2 進数 stu である．

$p\,q\,r$	$s\,t\,u$	
-00	$--0$	$(f_0+f_4)+(f_2+f_6)$
-01	$--0$	$(f_0+f_4\omega^4)+(f_2+f_6\omega^4)\omega^2$
-10	$--0$	$(f_0+f_4)+(f_2+f_6)\omega^4$
-11	$--0$	$(f_0+f_4\omega^4)+(f_2+f_6\omega^4)\omega^2\omega^4$
-00	$--1$	$(f_1+f_5)+(f_3+f_7)$
-01	$--1$	$(f_1+f_5\omega^4)+(f_3+f_7\omega^4)\omega^2$
-10	$--1$	$(f_1+f_5)+(f_3+f_7)\omega^4$
-11	$--1$	$(f_1+f_5\omega^4)+(f_3+f_7\omega^4)\omega^2\omega^4$

る．第 2 列は h の 2 進数 stu である．

表 10.3 は，式 (10.25) の最後の括弧内の項である第 1 段階の変換 $\displaystyle\sum_{s=0}^{1}\omega^{4rs}f_{stu}$ の処理が行われた後の状況である．この変換を行うと，h のインデックス s は消えて tu だけになり，k のインデックスの r が現れる．まず，それぞれ表 10.2 の第 2 行と第 6 行を組み合わせ，$r=0$ の場合は，第 2 行と第 6 行の和 $f_{0tu}+f_{1tu}$，$r=1$ の場合は差 $f_{0tu}-f_{1tu}\omega^4=f_{0tu}+f_{1tu}\omega^4$ となり，表 10.3 の第 2 行，第 3 行とする．行の組み合わせは $(2,6)\to(2,3)$，$(3,7)\to(4,5)$，$(4,8)\to(6,7)$，$(5,9)\to(8,9)$ である．

表 10.4 は第 2 段階の変換 $\displaystyle\sum_{t=0}^{1}\omega^{(4q+2r)t}\cdots$ である．この変換を行うと，h のインデックス t も消えて u だけになり，k のインデックスは qr になる．今回の行の組み合わせは $(2,6)\to(2,4)$，$(3,7)\to(3,5)$，$(4,8)\to(6,8)$，$(5,9)\to(7,9)$ である．

表 10.5 は第 3 段階の変換 $\displaystyle\sum_{u=0}^{1}\omega^{(4p+2q+r)u}\cdots$ である．この変換を行うと，h のイン

表 10.5　FFT の第 3 段階の変換．第 1 列は k の 2 進数 pqr である．h の 2 進数 stu はなくなっている．

pqr	
000	$(f_0+f_4)+(f_2+f_6)+((f_1+f_5)+(f_3+f_7))$
001	$(f_0+f_4\omega^4)+(f_2+f_6\omega^4)\omega^2+((f_1+f_5\omega^4)+(f_3+f_7\omega^4)\omega^2)\omega^1$
010	$(f_0+f_4)+(f_2+f_6)\omega^4+((f_1+f_5)+(f_3+f_7)\omega^4)\omega^2$
011	$(f_0+f_4\omega^4)+(f_2+f_6\omega^4)\omega^2\omega^4+((f_1+f_5\omega^4)+(f_3+f_7\omega^4)\omega^2\omega^4)\omega^3$
100	$(f_0+f_4)+(f_2+f_6)+((f_1+f_5)+(f_3+f_7))\omega^4$
101	$(f_0+f_4\omega^4)+(f_2+f_6\omega^4)\omega^2+((f_1+f_5\omega^4)+(f_3+f_7\omega^4)\omega^2)\omega^1\omega^4$
110	$(f_0+f_4)+(f_2+f_6)\omega^4+((f_1+f_5)+(f_3+f_7)\omega^4)\omega^2\omega^4$
111	$(f_0+f_4\omega^4)+(f_2+f_6\omega^4)\omega^2\omega^4+((f_1+f_5\omega^4)+(f_3+f_7\omega^4)\omega^2\omega^4)\omega^3\omega^4$

デックス u も消え，k のインデックスは pqr となり，変換が終了する．今回の行の組み合わせは $(2,6)\to(2,6)$，$(3,7)\to(3,7)$，$(4,8)\to(4,8)$，$(5,9)\to(5,9)$ である．表 10.5 を整理すれば，式 (10.16) であることがわかる．

プログラム (Fast_Fourier_Transform.java) のコンストラクタ (Fast_Fourier_Transform) は以下のとおりである．

Fast_Fourier_Transform.java のコンストラクタ (Fast_Fourier_Transform)：

```
1   public Fast_Fourier_Transform(int n, double xwidth) {
2     double rm= Math.log((double)n)/Math.log(2.0);
3     if (Math.pow(2.0,rm)!=n) {
4       System.out.println("ERROR! n is not 2^m.");
5       System.exit(0);
6     }
7     m= (int)(rm+0.1);
8     this.n= n;
9     n2= n/2;
10    this.xwidth= xwidth;
11    ra= new double[n];
12    ia= new double[n];
13    rb= new double[n];
14    ib= new double[n];
15    rom= new double[n];
16    iom= new double[n];
17  }
```

1. L1〜17: コンストラクタ Fast_Fourier_Transform の定義である．引数の n はデータの数であり m を整数として 2^{m} でなければならない．xwidth は変換する x 座標の幅である．
2. L2〜7: データの数 n から m を定めている．n が 2^{m} でなかったら警告を出力して，処理を止める．
3. L11, 12: ra, ia は式 (10.13) のデータ f の実数部と虚数部のための配列である．

4. L13, 14: rb, ib は作業用の実数部と虚数部のための配列である．
5. L15, 16: rom, iom は変換に使う式 (10.13) の ω の実数部と虚数部の配列である．

プログラム (Fast_Fourier_Transform.java) のメソッド (fast_Fourier_Transform) と，その説明は以下のとおりである．

Fast_Fourier_Transform.java のメソッド (fast_Fourier_Transform)：

```
 1  public double[][] fast_Fourier_Transform(double[][] f, int is) {
 2  中略
 3    for (i=0; i<n; i++) {
 4  中略
 5    }
 6    iom[  0]= 0.0;
 7  中略
 8    for (int l=0; l<m; l++) {
 9  中略
10    }
11    s= xwidth/n/Math.sqrt(2*Math.PI);
12    for (i=0; i<n; i++) {
13  中略
14    }
15    return ans;
16  }
```

1. L1〜16: メソッド fast_Fourier_Transform の定義である．引数の f はデータの配列 double[2][n] であり，実数部と虚数部が代入されている．is は正でフーリエ変換，負で逆フーリエ変換であることを指定する．
2. L2: 作業用の変数や配列の定義がある．
3. L3〜5: f の値を ra, ia にコピーする．
4. L6, 7: ω のための sin, cos 関数の値を計算し，配列 rom, iom に代入する．
5. L8〜10: 式 (10.25) を使って，変換を繰り返し行う．
6. L11: 変換後に掛ける係数を作る．
7. L12〜14: 変換後の実数部と虚数部に係数を掛け，配列 ans に代入する．
8. L15: 戻り値 ans を返す．

10.4　3 次元極座標系におけるフーリエ変換

2 次元あるいは 3 次元直交座標系におけるフーリエ変換は，1 次元のフーリエ変換を各次元について行えばよい．3 次元極座標系におけるフーリエ変換は次のようになる．

3 次元極座標系は図 10.5 である．r は原点からの距離で動径といわれ，θ は z 軸からの角度で天頂角，ϕ は x 軸からの角度で偏角 (方位角) といわれる．
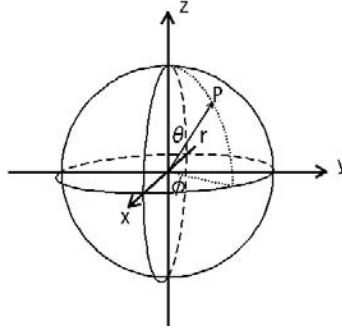
図 10.5　3 次元極座標系．r は動径，θ は天頂角，ϕ は方位角である．

点 P の座標は，直交座標系では (x, y, z)，3 次元極座標系では (r, θ, ϕ) であり，3 次元極座標系と直交座標系の関係は

$$\begin{aligned} x &= r\sin\theta\cos\phi \\ y &= r\sin\theta\sin\phi \\ z &= r\cos\theta \end{aligned} \tag{10.26}$$

である．関数を 3 次元極座標系で表す場合には，動径座標 r のみの動径関数 $R_l(r)$ と角座標 θ, ϕ の関数である球面調和関数 $Y_{lm}(\theta, \phi)$ の積での展開

$$\psi(\mathbf{r}) = \sum_{l=0}^{\infty}\sum_{m=-l}^{l} i^l a_{lm} R_l(r) Y_{lm}(\theta, \phi) \tag{10.27}$$

となる．l は負でない整数（$l \geq 0$）であり，m は $-l$ から l までの $2l+1$ 個の整数の値（$-l \leq m \leq l$）をとれる．球面調和関数は具体的には次の式で定義される．

$$Y_{lm}(\theta, \phi) = (-1)^m \sqrt{\frac{2l+1}{4\pi}\frac{(l-m)!}{(l+m)!}} P_l^m(\cos\theta) e^{im\phi} \tag{10.28}$$

ここで，$P_l^m(\cos\theta)$ はルジャンドル陪関数である．これらの球面調和関数は完全直交規格関数系をなしているので，直交規格関数系として

$$\int_0^{2\pi}\int_0^{\pi} Y_{lm}^*(\theta, \phi) Y_{l'm'}(\theta, \phi)\sin\theta \mathrm{d}\theta \mathrm{d}\phi = \delta_{ll'}\delta_{mm'} \tag{10.29}$$

が成り立つ．また完全系として

$$\sum_{lm} Y_{lm}^*(\theta, \phi) Y_{lm}(\theta', \phi') = \delta(\theta-\theta')\delta(\phi-\phi') \tag{10.30}$$

が成り立つ．$\delta_{nn'}$ はクロネッカーのデルタであり，$\delta(x)$ はディラックのデルタ関数である．球面調和関数の初めの $l = 0, 1, 2$ を具体的に記すと次のとおりである．

$$
\left.
\begin{aligned}
Y_{00} &= \frac{1}{\sqrt{4\pi}} \\
Y_{1-1} &= \sqrt{\frac{3}{8\pi}}\sin\theta e^{-i\phi} \\
Y_{10} &= \sqrt{\frac{3}{4\pi}}\cos\theta \\
Y_{11} &= -\sqrt{\frac{3}{8\pi}}\sin\theta e^{i\phi} \\
Y_{2-2} &= \sqrt{\frac{15}{32\pi}}\sin^2\theta e^{-2i\phi} \\
Y_{2-1} &= \sqrt{\frac{15}{8\pi}}\sin\theta\cos\theta e^{-i\phi} \\
Y_{20} &= \sqrt{\frac{5}{16\pi}}(3\cos^2\theta - 1) \\
Y_{21} &= -\sqrt{\frac{15}{8\pi}}\sin\theta\cos\theta e^{i\phi} \\
Y_{22} &= \sqrt{\frac{15}{32\pi}}\sin^2\theta e^{2i\phi}
\end{aligned}
\right\}
\tag{10.31}
$$

関数 $\psi(\mathbf{r})$ のフーリエ変換は

$$
\chi(\mathbf{q}) = \frac{1}{\sqrt{8\pi^3}}\int\!\!\int\!\!\int \psi(\mathbf{r})e^{-i\mathbf{qr}}\mathrm{d}^3\mathbf{r} \tag{10.32}
$$

である．球面調和関数と球ベッセル関数 $j_l(qr)$ を用いた指数関数の部分波展開の公式

$$
e^{-i\mathbf{q}\cdot\mathbf{r}} = 4\pi\sum_{l=0}^{\infty}\sum_{m=-l}^{l}(-i)^l Y_{lm}(\theta_q,\,\phi_q)j_l(qr)Y_{lm}^*(\theta_r,\,\phi_r) \tag{10.33}
$$

を使うと，式 (10.32) は

$$
\begin{aligned}
\chi(\mathbf{q}) &= \sqrt{\frac{2}{\pi}}\int\!\!\int\!\!\int\sum_{lm}\sum_{l'm'} i^l a_{lm}R_l(r)Y_{lm}(\theta_r,\,\phi_r) \\
&\quad (-i)^{l'}Y_{l'm'}(\theta_q,\,\phi_q)j_{l'}(qr)Y_{l'm'}^*(\theta_r,\,\phi_r)r^2\sin\theta_r\mathrm{d}r\mathrm{d}\theta_r\mathrm{d}\phi_r \\
&= \sqrt{\frac{2}{\pi}}\sum_{lm}\sum_{l'm'} i^{l-l'}a_{lm}Y_{l'm'}(\theta_q,\,\phi_q)\int_0^{\infty}R_l(r)j_{l'}(qr)r^2\mathrm{d}r \\
&\quad \int\!\!\int Y_{lm}(\theta_r,\,\phi_r)Y_{l'm'}^*(\theta_r,\,\phi_r)\sin\theta_r\mathrm{d}\theta_r\mathrm{d}\phi_r \\
&= \sqrt{\frac{2}{\pi}}\sum_{lm}a_{lm}Y_{lm}(\theta_q,\,\phi_q)\int_0^{\infty}R_l(r)j_l(qr)r^2\mathrm{d}r \\
&= \sum_{lm}a_{lm}Y_{lm}(\theta_q,\,\phi_q)F_l(q)
\end{aligned}
\tag{10.34}
$$

となる．直交座標系 $x,\,y,\,z$ から，極座標系 $r,\,\theta,\,\phi$ に移ると，積分要素は $\mathrm{d}x\mathrm{d}y\mathrm{d}z = \dfrac{\partial(x,\,y,\,z)}{\partial(r,\,\theta,\,\phi)}\mathrm{d}r\mathrm{d}\theta\mathrm{d}\phi = r^2\sin\theta\mathrm{d}r\mathrm{d}\theta\mathrm{d}\phi$ に変わるが，図 10.6 はその直観的な説明である．
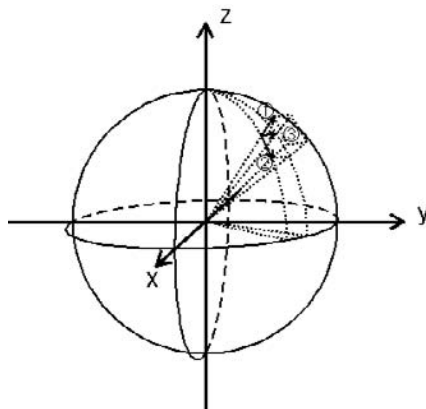
図 10.6 3 次元極座標系の積分要素．動径方向の増加は $\mathrm{d}r$ ①，天頂角の方向は子午線に沿って $r\mathrm{d}\theta$ ②，方位角の方向は緯線に沿って $r\sin\theta\mathrm{d}\phi$ ③ である．

図 10.5 の点 P から 3 本のベクトル $\mathrm{d}r,\ r\mathrm{d}\theta,\ r\sin\theta\mathrm{d}\phi$ が描かれている．これらの 3 本のベクトルで作られる近似的な直方体の体積が $r^2\sin\theta\mathrm{d}r\mathrm{d}\theta\mathrm{d}\phi$ となる．

式 (10.34) の $F_l(q)$ は動径フーリエ変換

$$F_l(q) = \sqrt{\frac{2}{\pi}}\int_0^\infty R_l(r)j_l(qr)r^2\mathrm{d}r \tag{10.35}$$

である．

図 10.7 は水素原子の $1s$ 動径波動関数のフーリエ変換，動径運動量波動関数 $F_{1s}(q)$ を式 (10.35) を使って求め，$F_{1s}(q)$ に動径座標 q を掛けた $qF_{1s}(q)$ である．実線は解析的にわかっている $1s$ 動径波動関数 $R_{1s}(r) = 2e^{-r}$ から求めたものであり，∘ 印は図 6.4 (p.106) に示されている値を使って求めたものである．図 6.4 の関数は $P_{1s}(r) = rR_{1s}(r)$ である．当然なことであるが，両者の動径運動量波動関数は一致している．また，

$$\int_0^{10}|F_{1s}(q)|^2q^2\mathrm{d}q = 0.99998 \tag{10.36}$$
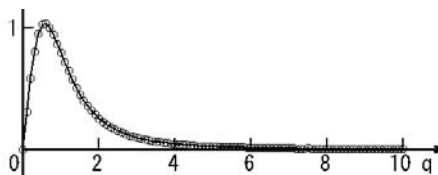
図 10.7 水素原子の $1s$ 動径運動量波動関数 $qF_{1s}(q)$．q は運動量空間の動径座標であり，単位は $\dfrac{2\pi}{\mathrm{a.u.}}$ である．a.u. は原子単位を表している．実線は解析的な $1s$ 動径波動関数を解析的にフーリエ変換した関数であり，∘ 印は図 6.4 で使われている数値をフーリエ変換して求めたものである．

であるから規格化されており，$q = 0 \sim 10$ で 0.99998 になっている．

解析的な 1s 動径波動関数 $R_{1s}(r) = 2e^{-r}$ のフーリエ変換は，以下のとおりである．$R_{1s}(r) = 2e^{-r}$ を式 (10.35) に当てはめると

$$\begin{aligned} F_{1s}(q) &= \sqrt{\frac{2}{\pi}}\int_0^\infty R_{1s}(r)j_0(qr)r^2\mathrm{d}r = \sqrt{\frac{2}{\pi}}\int_0^\infty 2e^{-r}\frac{\sin(qr)}{qr}r^2\mathrm{d}r \\ &= \sqrt{\frac{2}{\pi}}\frac{2}{q}\int_0^\infty e^{-r}\sin(qr)r\mathrm{d}r \end{aligned} \tag{10.37}$$

となる．不定積分の公式

$$\begin{aligned} \int e^{-r}\sin(qr)\mathrm{d}r &= \frac{e^{-r}}{1+q^2}(-\sin(qr) - q\cos(qr)) \\ \int e^{-r}\cos(qr)\mathrm{d}r &= \frac{e^{-r}}{1+q^2}(-\cos(qr) + q\sin(qr)) \end{aligned} \tag{10.38}$$

を使って部分積分をすると，式 (10.37) は

$$F_l(q) = \sqrt{\frac{2}{\pi}}\frac{2}{q}\frac{1}{1+q^2}\int_0^\infty \left(e^{-r}\sin(qr) + qe^{-r}\cos(qr)\right)\mathrm{d}r = \sqrt{\frac{2}{\pi}}\frac{4}{(1+q^2)^2} \tag{10.39}$$

となる．図 10.7 の実線はこの関数 $qF_{1s}(q)$ を表示したものである．

10.5　畳み込みと畳み込み除去

すべての実験で実験誤差が入ることを避けるのは不可能である．一般的に，ある変数 x についての分散を求める実験の際に，理論値は $f(x)$ であるのに，実験を行うと $f(x) + \epsilon$ の形で生じる誤差もあるし，$f(x + \epsilon)$ の形で生じる誤差もあり，また，両者が同時に生じることも多い．前者は，現象の回数が少ないために生じるとも考えられ，統計誤差といわれる．この誤差の補正としては，データの平滑化が使われる．後者は，例えば，光学系の場合に光源が点光源ではなかったり，検出器の口径に大きさがあり，点でないためなどで生じる，いわゆる「ぼけ」である．このほか，エネルギー分散が得られる実験では，現象の時間が有限であるために，不確定性関係から生じるばらつきもある．この場合は，理論値を「ぼけ」の原因となる関数を使ってぼかして (畳み込み，convolution) 実験値と比較検討したり，実験値に畳み込み除去 (deconvolution) を施して，理論値と比較したりする場合もある．

畳み込みを起こす関数としては，ガウス関数が当てはまる場合と，ローレンツ関数 (コーシー関数) が当てはまる場合などがある．$(-\infty, \infty)$ の領域で規格化されているこれらの関数型は，それぞれ

$$\begin{aligned} g(x) &= \frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{x^2}{2\sigma^2}} \quad \text{(ガウス関数)} \\ g(x) &= \frac{\Gamma}{2\pi}\frac{1}{x^2 + (\frac{\Gamma}{2})^2} \quad \text{(ローレンツ関数)} \end{aligned} \tag{10.40}$$

である．ガウス関数の σ は標準偏差であり，ローレンツ関数の Γ は半値幅である．ともに値が大きくなると分布が広がる．

10.5.1　畳み込み

図 10.8 に，フーリエ変換の応用である畳み込みの一例として，金属銅による X 線の吸収曲線の実験値 $h(E)$ と理論値 $f(E)$，および理論値にローレンツ関数で畳み込みを施したものを示す．

理論値が，関数 $f(x)$ で与えられる場合に，それをぼかす原因となる実験系の関数が $g(x)$ であれば，点 x' での理論値 $f(x')$ はぼけて $g(x-x')f(x')$ となる．そこで，実験で得られる関数は

$$h(x) = \int g(x-x')f(x')\mathrm{d}x' \tag{10.41}$$

となる．もちろん，各 x についてこの積分を直接行うこともあるが，次のように，フーリエ変換を使って求めることもできる．式 (10.41) の両辺のフーリエ変換を行うと，左辺は

$$H(q) = \int h(x)e^{-iqx}\mathrm{d}x \tag{10.42}$$

となり，右辺は
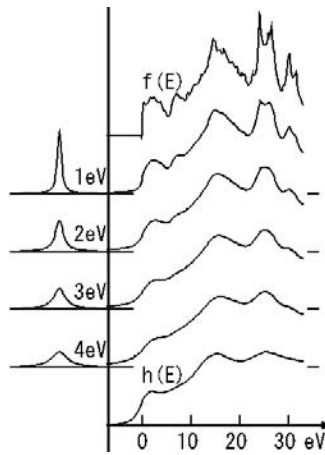
図 10.8　金属銅による X 線吸収曲線理論値の畳み込み．最上段が理論値 $f(E)$，最下段が実験値 $h(E)$ であり，他は，理論値にローレンツ関数で畳み込みを施した後の値である．横軸の単位は電子ボルト（eV）であり，原点は吸収端（フェルミ準位）である．縦軸の単位は不定であるが，実験と理論で面積がほぼ等しくなるように規格化してある．上から 2 段目の左は，$\Gamma = 1\mathrm{eV}$ としたローレンツ関数であり，右はそれを使って理論値に畳み込みを施したものである．以下順に $\Gamma = 2, 3, 4\,\mathrm{eV}$ の場合である．（Z10_08_conv_Cu.java）

$$\begin{aligned}&\iint g(x-x')f(x')\mathrm{d}x' e^{-iqx}\mathrm{d}x\\&\quad=\int g(x-x')e^{-iq(x-x')}\mathrm{d}x\int f(x')e^{-iqx'}\mathrm{d}x'\\&\quad=G(q)F(q)\end{aligned}\tag{10.43}$$

となるので，

$$H(q)=G(q)F(q)\tag{10.44}$$

が成り立つ．

図 10.9 は，図 10.8 に示されている理論値，実験値，それと $\Gamma = 1, 2, 3, 4\,\mathrm{eV}$ とした 4 個のローレンツ関数のフーリエ変換である．

ここで用いたフーリエ変換プログラムは FFT である．実験データは図 10.8 に示されている範囲で，等分割の 256 点で得られている．理論値は実験値にならって計算されている．このままのデータ領域でフーリエ変換を行うと，q 座標の範囲の幅は約 $40\dfrac{2\pi}{\mathrm{eV}}$ であるから，q 座標での点のとり方が極端に粗いことになり，$1\dfrac{1}{\mathrm{eV}}$ 当たり約 6 点しかないので，x 座標を広げなければならない．まず，高エネルギー端での大きなステップの影響を抑えるため，高エネルギー端でデータを左右反転したものをつなげ，左右対称のデータとした．さらにデータの両側にはゼロのデータを補い，データ領域を $640\,\mathrm{eV}$ に広げ，データ点を $4,096$ 点とした．q 座標の幅は約 $40\dfrac{1}{\mathrm{eV}}$ であるが，図には $5\dfrac{1}{\mathrm{eV}}$ までしか示していない．理論値と実験値を比較すると，0.5 あたりまでは似通っているが，それ以後は，実験値は収束していくが理論値は振動が続き $f(x)$ が滑らかでないことを反映している．4 個のローレンツ関数 $g(x)$ は図 10.8 に示されている
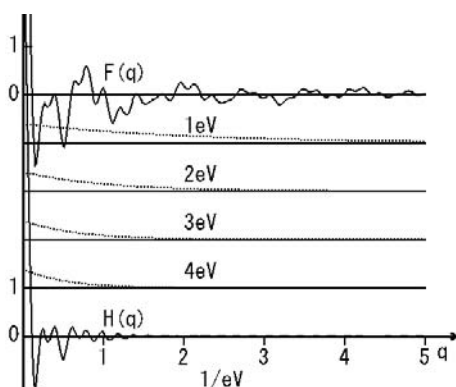
図 10.9　金属銅による X 線吸収曲線の理論値，実験値，それと 4 個のローレンツ関数のフーリエ変換．最上段が理論値 $F(q)$，最下段が実験値 $H(q)$ である．上から 2 段目は，$\Gamma = 1\,\mathrm{eV}$ としたローレンツ関数 $G(q)$ であり，以下順に $\Gamma = 2, 3, 4\,\mathrm{eV}$ の場合である．横軸の単位は 1/eV であり，縦軸については理論値と実験値は不定である．ローレンツ関数については，原点での値が $1/\pi = 0.3183\cdots$ である．（Z10_09_fourier_Cu.java）

ように，$\Gamma = 1, 2, 3, 4\,\mathrm{eV}$ の順に幅が広くなっている．それを反映して，図 10.9 では $G(q)$ の広がりがこの順に狭くなっている．

図 10.8 には，式 (10.44) の逆フーリエ変換によって畳み込みを行った結果が，$\Gamma = 1, 2, 3, 4\,\mathrm{eV}$ の順に示されており，徐々にぼけ具合が激しくなっている．ちなみに，この実験での実験系の Γ は約 $2\,\mathrm{eV}$ である．

プログラム (Z10_08_conv_Cu.java) のメインメソッドの一部と，その説明は以下のとおりである．

Z10_08_conv_Cu.java のメインメソッドの一部：

```
 1     double[] hrd= {
 2   中略
 3     };
 4     double[] frd= {
 5   中略
 6     } ;
 7     int nd= frd.length;
 8     double dwidth= 40.0;
 9     double de= dwidth/nd;
10     int n= 4096;
11     int n2= n/2;
12     double emin= -n2*de;
13     double emax= n2*de;
14   中略
15     double[] e= new double[n];
16     double[][] hr= new double[2][n];
17     double[][] hf= new double[2][n];
18     double[][] fr= new double[2][n];
19     double[][] ff= new double[2][n];
20     double[][] gr= new double[2][n];
21     double[][] gf= new double[2][n];
22     double[][] wr= new double[2][n];
23     double[][] wf= new double[2][n];
24     int i, j, k;
25     double qmax= n2*2*Math.PI/(emax-emin);
26     double qmin= -qmax;
27     for (i=0; i<n; i++) {
28       e[i]= emin+i*de;
29       hr[0][i]= 0.0;
30       hr[1][i]= 0.0;
31       fr[0][i]= 0.0;
32       fr[1][i]= 0.0;
33     }
34     int min= n2-nd;
35     for (i=min; i<n2; i++) {
36       j= n-i-1;
37       hr[0][i]= hr[0][j]= hrd[i-min];
38       fr[0][i]= fr[0][j]= frd[i-min];
39     }
```

```
40    double snorm= nd/2;
41    double sum= 0.0;
42    for (i=0; i<n; i++) {
43       sum= sum+hr[0][i];
44    }
45    sum= snorm/sum;
46    for (i=0; i<n; i++) {
47      hr[0][i]= hr[0][i]*sum;
48    }
49    sum= 0.0;
50    for (i=0; i<n; i++) {
51       sum= sum+fr[0][i];
52    }
53    sum= snorm/sum;
54    for (i=0; i<n; i++) {
55      fr[0][i]= fr[0][i]*sum;
56    }
57    Fast_Fourier_Transform ffc= new Fast_Fourier_Transform(n, (emax-emin));
58    Fast_Fourier_Transform ffcq= new Fast_Fourier_Transform(n, (qmax-qmin));
59  中略
60    gk.gpl(n2-nd,nd,e,hr[0]);
61    ff= ffc.fast_Fourier_Transform_C(fr, 1);
62    double x;
63    double[] gam= {1.0, 2.0, 3.0, 4.0};
64    String[] Stgam= {"1eV", "2eV", "3eV", "4eV"};
65    double gamma, c;
66    for (k=0; k<4; k++) {
67      gamma= gam[k];
68      c= gamma/(2*Math.PI);
69      for (i=0; i<n; i++) {
70        x= (n2-i)*de;
71        gr[0][i]= c/(x*x+gamma*gamma/4);
72        gr[1][i]= 0.0;
73       }
74      gf= ffc.fast_Fourier_Transform_C(gr, 1);
75  中略
76      for (i=0; i<n; i++) {
77        wf[0][i]= ff[0][i]*(gf[0][i]*Math.sqrt(2*Math.PI));
78        wf[1][i]= 0.0;
79      }
80      wr= ffcq.fast_Fourier_Transform_C(wf, -1);
81  中略
82      gk.gpl(n2-nd,nd,e,wr[0]);
83  中略
84    }
85  中略
86    gk.gpl(n2-nd,nd,e,fr[0]);
```

1. L1〜3: 実験データの配列.
2. L4〜6: 理論値の配列.

3. L7〜9: nd は理論値および実験データの個数，dwidth はエネルギー幅 40 eV，de はエネルギー刻みである．
4. L10, 11: FFT で扱うデータ数を 4,096 とする．
5. L12, 13: エネルギー幅を FFT で扱うデータ数に合わせて emin から emax までに広げる．
6. L15: エネルギー座標の配列．
7. L16〜23: フーリエ変換で使う配列．実験値，理論値，ローレンツ関数，作業用の配列を，それぞれ e 空間， q 空間用の二つずつ定義している．各配列には，実数部と虚数部がある．
8. L25, 26: q 座標の範囲を指定する．
9. L27〜33: e 空間座標値の設定とデータ用配列をゼロで初期化.
10. L34〜39: データ値を中央部に代入する．負側の左から中央まではデータを昇順にコピーし，右側には左右を反転して降順にコピーする．
11. L40〜48: 実験データを規格化する．
12. L49〜56: 理論データを規格化する．
13. L57: Fast_Fourier_Transform のインスタンス ffc を作る．
14. L58: Fast_Fourier_Transform のインスタンス ffcq を作る．
15. L60: 実験値データのグラフを描く．
16. L61: 実験値データのフーリエ変換を配列 ff に与える．
17. L63: ローレンツ関数の Γ 値を与える．
18. L66〜84: 4 種のローレンツ関数のループをまわす．
19. L68: ローレンツ関数の係数を作る．
20. L69〜73: ローレンツ関数の値を配列に作る．
21. L74: ローレンツ関数のフーリエ変換を配列 gf に与える．
22. L76〜79: 理論値とローレンツ関数のフーリエ変換値の積を作る．両関数ともに偶関数なので，虚数部はゼロである．
23. L80: フーリエ逆変換で畳み込みを行う．
24. L82: 畳み込み後のデータのグラフを描く．
25. L84: 4 種のローレンツ関数のループの終了．
26. L86: 理論値データのグラフを描く．

10.5.2 畳み込み除去

式 (10.44) を変形した

$$F(q) = \frac{H(q)}{G(q)} \tag{10.45}$$

の逆フーリエ変換を行えば，原理的には「ぼけ」を除去した関数である $f(x)$ を得ることができることになるが，分母にある $G(q)$ の値が小さくなると $F(q)$ に発散が生じてしまう．そこで，適当な q より大きいところでは，$F(q)$ を小さくする cut off をしなければならないので，結果は厳密なものではあり得ないといえる．図 10.10 には，金属銅による X 線吸収曲線の理論値 $f(x)$ と実験値 $h(x)$ のほか，式 (10.45) の逆フーリエ変換によって畳み込み除去を行った結果が，$\Gamma = 1, 2, 3, 4\,\mathrm{eV}$ の順に示されている．cut off は約 $1.25\dfrac{1}{\mathrm{eV}}$ としている．徐々に特徴がはっきりしてきてはいるが，Γ の値が
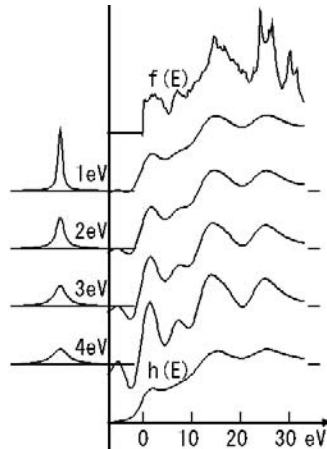
図 10.10　金属銅による X 線吸収曲線実験値の畳み込み除去．最下段は実験値 $h(E)$，最上段は理論値 $f(E)$ であり，以下順に実験値を $\Gamma = 1, 2, 3,$ 4eV のローレンツ関数で畳み込みの除去を施した後の値である．横軸の単位は電子ボルト（eV）であり，原点は吸収端 (フェルミ準位) である．（Z10_10_deconv_Cu.java）

3 eV 以上になると，余計な振動が激しくなり，理論値がなければどこまでが真実であるかの判定は難しいかもしれない．ここでは，cut off を step function としているが，滑らかにゼロにするなどの工夫も必要であろう．

プログラム (Z10_10_deconv_Cu.java) のメインメソッドの一部と，その説明は以下のとおりである．

Z10_10_deconv_Cu.java のメインメソッドの一部：

```
 1     int icut= n2/16;
 2     for (k=0; k<4; k++) {
 3   中略
 4       gf= ffc.fast_Fourier_Transform_C(gr, 1);
 5   中略
 6       for (i=0; i<n; i++) {
 7         if (i<n2-icut || i>n2+icut) {
 8           wf[0][i]= 0.0;
 9         } else {
10           wf[0][i]= hf[0][i]/(gf[0][i]*Math.sqrt(2*Math.PI));
11         }
12         wf[1][i]= 0.0;
13       }
14   中略
15        wr= ffcq.fast_Fourier_Transform_C(wf, -1);
16     }
```

1. L1: 高周波部分をカットするための位置．場合によっては，カットを滑らかにする必要があり，工夫の余地がある．

2. L2〜15: 4 種類のローレンツ関数のループをまわす．
3. L4: ローレンツ関数のフーリエ変換が配列 gf に与えられる．
4. L6〜13: 実験値の変換後の値を，ローレンツ関数の変換後の値で割り，配列 wf に代入する．この際，カットが必要であり，ここでは単純に中心から icut の両外側をゼロにしているだけである．
5. L12: 偶関数であるから，虚数部はゼロである．
6. L15: 配列 wf の逆フーリエ変換により，畳み込み除去を行う．

演習問題

10.1　ガウス関数のフーリエ変換プログラム Z10_03_gauss.java を変更して，関数 $\dfrac{\sin gx}{\pi x}$ のフーリエ変換プログラム Q10_01_Z10_03_delta.java を作りなさい．ただし，$g=\pi/2,\ \pi,\ 3\pi/2$ とすること．ちなみに，$\delta(x)=\lim_{g\to\infty}\dfrac{\sin gx}{\pi x}$ は式 (10.30) に出てきたデルタ関数である．

解答例： ソースプログラム Q10_01_Z10_03_delta.java

$$\int_{-\infty}^{\infty}\frac{\sin gx}{\pi x}\cos qx\mathrm{d}x=\begin{cases}1 & (|q|<g)\\ 1/2 & (|q|=g)\\ 0 & (|q|>g)\end{cases}$$

であるが，積分領域が有限であるため振動が生じている．

第 11 章

乱数

最近のプログラミング言語の多くには，$0 \leq x < 1$ の範囲に一様に分布する擬似乱数を返す関数が用意されているので，乱数発生のアルゴリズムは省き，その利用についていくつかの例を述べる．

11.1 モンテカルロ法と積分

図 11.1 は，$-1 \leq x < 1$ の範囲で乱数を作り x 座標値とし，続いて $-1 \leq y < 1$ の範囲で乱数を作って y 座標値として，$-1 \leq x < 1$ と $-1 \leq y < 1$ の正方形内に点を打っていったものである．それらの点のうち，半径 1 の円の中に入る点の数を数えると，全体の点の数に対するその数の比に正方形の面積 4 を掛けたものが，円の面積 (円周率) になる．

プログラム (Z11_01_monte.java) のメソッドの一部と，その説明は以下のとおりである．
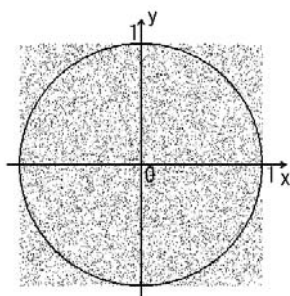
図 11.1　ヒット・オア・ミス法による円周率．2 個の乱数を一組として作った座標値 (x, y) をプロットした．点の数は 10,000 点である．半径 1 の円の内部にプロットされる点の数の割合は，ほぼ $\frac{\pi}{4}$ になる．(Z11_01_monte.java)

Z11_01_monte.java のメソッドの一部：

```
1    double x,y;
2    int in=0;
3    int max=10000;
4    for (int i=0; i<max; i++) {
5      x= 2*Math.random()-1.0;
6      y= 2*Math.random()-1.0;
7      gk.gdot(x,y);
8      if (x*x+y*y <= 1.0) in++ ;
9    }
```

1. L1: $[-1, 1)$ の乱数を代入する変数である．
2. L2: 円の内側にヒットした点の数を数える変数であり，ゼロに初期化しておく．
3. L3: 発生させる点の数．
4. L4〜9: 発生させる点の数だけループをまわす．
5. L5, 6: メソッド Math.random は，範囲が $[0,\ 1)$ の一様分布乱数であるから，$[-1, 1)$ の範囲に座標変換して，変数 x，y に代入する．
6. L7: 座標 (x，y) に点を表示する (グラフィックスの準備はすでにしてある)．
7. L8: 点の位置の原点からの距離が 1 以下であったらヒットしたので，ヒットした点の数に 1 を加える．
8. L9: ループの終わりである．

表 11.1 は，このようにして求めた円周率の値と積分で求めた値が，使った乱数の数により，どのように収束していくかを示すものである．数 % の精度でよいなら 1,000 点程度でよいが，1% 以下に抑えようとすると数万点が必要となる．これは，n 回の試行を行うと誤差は $\sqrt{n}$ となり，相対誤差は $\dfrac{1}{\sqrt{n}}$ となることに対応している．

このように，円の内部という条件に「合うかどうか」を使う方法をヒット・オア・

表 11.1　乱数で求めた円周率．2 個の乱数を一組として作った座標 (x, y) のうち，半径 1 の円内の点の割合から求めた円周率 (ヒット・オア・ミス法) と乱数を使ったモンテカルロ法の積分 $2\int_{-1}^{1}\sqrt{1-x^2}\mathrm{d}t$ の値である．点の数は使った点の数，数値は得られた円周率の値である．

点の数	hit	積分	点の数	hit	積分
真値	3.1415		10000	3.150	3.140
100	3.36	3.19	15000	3.155	3.143
200	3.44	3.09	20000	3.148	3.144
400	3.27	3.12	25000	3.152	3.144
800	3.12	3.17	30000	3.152	3.144
1000	3.12	3.14	35000	3.149	3.145
2000	3.17	3.13	40000	3.142	3.146
4000	3.17	3.14	45000	3.145	3.146
8000	3.15	3.14	50000	3.147	3.145

ミス法という．この積分を乱数で求めるもう一つの方法は，範囲が $0 \le r < 1$ の乱数から，変換 $x = 2r - 1$ によって x の乱数を作り，積分を

$$2\int_{-1}^{1}\sqrt{1-x^2}\mathrm{d}t = \frac{1}{n}\sum_{i=1}^{n} f(x_i) \tag{11.1}$$

のように変形して，関数値の平均をとる方法である．ただし，この場合は $f(x) = 2\sqrt{1-x^2}$ である．ヒット・オア・ミス法と比較して，ほとんど優れてはいない．他方，同じ積分をガウス・ルジャンドルの 32 点で行った値は 3.14164 であり，ずっと優れている．しかし，次元の高い多重積分で 1〜2 桁の有効数字の答えを求める場合などには，モンテカルロ法の積分が有効である．

11.2　関数型分布乱数

先の $2\int_{-1}^{1}\sqrt{1-x^2}\mathrm{d}t$ のモンテカルロ法の積分では，変数 x の積分領域が $(-1, 1)$ であるから，区間 $(0, 1)$ の一様分布に従う確率変数である乱数 r から，変換 $x = 2r - 1$ によって x の乱数を作った．一般的に，一様分布ではあるが x の区間が (a, b) の場合には，変数変換をする変換関数 $x = a + (b - a)r$ を使えばよい．しかし，各種の分布関数 $f(x)$ に従う確率変数 x が必要な場合には，変換関数はそれほど単純ではない．

11.2.1　標準指数分布乱数

図 11.2 は次に述べる逆関数法で求めた標準指数分布型

$$f(x) = e^{-x} \tag{11.2}$$

の乱数である．x の刻みは 0.04 で，発生させた乱数の数は 10 万である．

この図の意味は，$[0, 1)$ の範囲に一様分布する r の値を変数変換して x にすると，ある x の値の出現頻度が，分布関数 $f(x)$ に比例するということである．その変換関数の作り方は以下のとおりである．

式 (11.2) の累積分布関数は

$$F(x) = \int^{x} f(t)\mathrm{d}t = \int_{0}^{x} e^{-t}\mathrm{d}t = 1 - e^{-x} \tag{11.3}$$
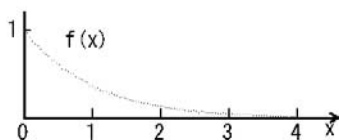
図 11.2　逆関数法で求めた標準指数分布型乱数の分布．x の刻みは 0.04 で，発生させた乱数の数は 10 万である．(Z11_02_exp.java)

であるから，この関数の下限値 $F(0) = 0$ を乱数 r の下限値 0 に対応させ，上限値 $F(\infty) = 1$ を r の上限値 1 に対応させれば，x の分布は指数分布になる．そこで標準指数分布型乱数は，乱数 r に対応する x の式 (11.3) を r に等しいと置いてその逆関数をとればよいから，

$$x = F^{-1}(r) = -\log(1-r) \tag{11.4}$$

となる．これが求める変換関数である．このように，分布関数 $f(x)$ の累積分布関数 $F(x)$ が解析的に求められる場合には，その逆関数 $x = F^{-1}(r)$ がその分布を与える乱数となる．

プログラム (Random_Exp.java) のメソッド (random_Exp) と，その説明は以下のとおりである．

Random_Exp.java のメソッド (random_Exp)：

```
1     public static double random_Exp() {
2         return -Math.log(Math.random());
3     }
```

1. L1〜3: メソッド random_Exp の定義である．
2. L2: 式 (11.4) であるが，メソッド Math.random が [0, 1) の一様分布乱数なので，Math.log の引数は $1-r$ と同等な r にしてある．

図 11.3 の下図は分布関数 $f(x) = e^{-x}$ であり，上図は累積分布関数 $F(x) = 1 - e^{-x}$ である．そこで，発生させた乱数 r の値を上図の縦軸にとり，グラフ $F(x) = 1 - e^{-x}$ で対応させる x の値が標準指数分布型乱数の値となる．当然のことであるが，r が一様に分布しているので，対応する x は値の小さいところに多く分布している．
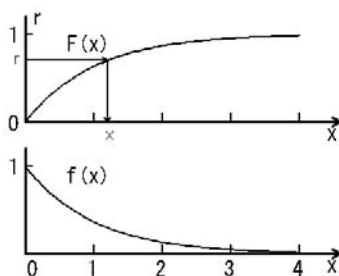
図 11.3　分布関数 $f(x) = e^{-x}$ と累積分布関数 $F(x) = 1 - e^{-x}$．上の図の縦軸上の r は発生させた乱数であり，$r = F(x)$ で対応する $x = F^{-1}(r)$ が標準指数分布の乱数となる．

11.2.2　ローレンツ型分布乱数

ローレンツ型分布関数の一般型は

$$f(x) = \frac{\Gamma}{2\pi}\frac{1}{x^2 + (\Gamma/2)^2} \tag{11.5}$$

である．$x = \pm\frac{\Gamma}{2}$ のときの関数の値は $f\left(\pm\frac{\Gamma}{2}\right) = \frac{f(0)}{2}$ であるから，Γ は関数値が原点での値の半分となる幅であり半値幅といわれる．$\Gamma = 1$ の場合の式 (11.5) の累積分布関数は

$$F(x) = \int_{-\infty}^{x} \frac{1}{\pi}\frac{1}{t^2+1}\mathrm{d}t = \frac{1}{\pi}\tan^{-1} x + 0.5 \tag{11.6}$$

であるから，この関数の下限値 $F(-\infty) = 0$ を乱数 r の下限値 0 に，上限値 $F(\infty) = 1$ を r の上限値 1 に対応させて線形関係を作れば，x の分布はローレンツ型分布になる．そこでローレンツ型分布乱数は，式 (11.6) の逆関数をとればよいから，

$$x = F^{-1}(r) = \tan(r - 0.5)\pi \tag{11.7}$$

となる．図 11.4 はローレンツ型分布乱数の分布である．x の刻みは 0.12 で，発生させた乱数の数は 10 万である．

プログラム (Random_Lorentzian.java) のメソッド (random_Lorentzian) と，その説明は以下のとおりである．

Random_Lorentzian.java のメソッド (random_Lorentzian)：

```
1     public static double random_Lorentzian() {
2         return Math.tan((Math.random()-0.5)*Math.PI);
3     }
```

1. L1〜3: メソッド random_Lorentzian の定義である．
2. L2: 式 (11.7) である．
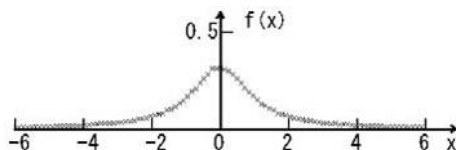
図 11.4　逆関数法で求めたローレンツ型分布乱数の分布．x の刻みは 0.12 で，発生させた乱数の数は 10 万である．（Z11_04_lorentzian.java）

11.2.3　正規分布乱数

正規分布関数は，平均が 0 で標準偏差を σ とすると

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-x^2/2\sigma^2} \tag{11.8}$$

である．累積分布関数は

$$\begin{aligned} F(x) &= \int_{-\infty}^{x} f(t)\mathrm{d}t = 0.5 + \int_{0}^{x} f(t)\mathrm{d}t \\ &= \frac{1}{2} + \frac{e^{-x^2/2\sigma^2}}{\sqrt{2\pi}} \sum_{i=0}^{\infty} \frac{(x/\sigma)^{2i+1}}{1 \cdot 3 \cdots (2i+1)} \end{aligned} \tag{11.9}$$

であるから，このままで逆関数を作ることはできない．そこで，解析的に累積分布関数の逆関数を作ることをやめ，数値積分で累積分布関数の表を作り，r に対応する値を参照して x とする表読み取り法を使えば，原理的にはあらゆる場合に逆関数法が使える．しかし，正規分布関数の場合には以下に述べるボックス・マーラー法がある．この方法では，累積分布関数が使える先の標準指数分布の式を使っており，極めて巧妙な方法である．標準指数分布関数 e^{-x} の定義域は $[0, \infty)$ であり，正規分布関数 e^{-x^2} の定義域は $(-\infty, \infty)$ であるから，そのままでは対応を付けられない．いま，

$$f(x, y) = e^{-(x^2+y^2)/2} \tag{11.10}$$

と置いて，これを y で積分すると

$$\begin{aligned} \int_{-\infty}^{\infty} f(x, y)\mathrm{d}y &= e^{-x^2/2} \int_{-\infty}^{\infty} e^{-y^2/2}\mathrm{d}y \\ &= \sqrt{2\pi} e^{-x^2/2} = 2\pi \frac{1}{\sqrt{2\pi}} e^{-x^2/2} = 2\pi f(x) \end{aligned} \tag{11.11}$$

となり，正規分布 $f(x)$ になる．同様に，$f(x, y)$ を x で積分すると正規分布 $f(y)$ になる．そこで，分布関数 $f(x, y) = e^{-(x^2+y^2)/2}$ の乱数を使えれば，y の値にはかかわらず x の値は正規分布乱数であり，x の値にはかかわらず y の値も正規分布乱数である．

次に

$$p = \frac{(x^2 + y^2)}{2} \tag{11.12}$$

と置くと，式 (11.10) は

$$f(p) = e^{-p} \tag{11.13}$$

となるから，標準指数分布の累積分布関数の式 (11.4) を使うことができる．具体的に正規分布乱数を作るには，1 個目の乱数 r_1 から標準指数分布乱数 p を作って動径座標に使い，2 個目の乱数 r_2 から範囲が $(0, 2\pi)$ の角座標値を作る．その値 (p, θ) を直交座標 (x, y) に変換すれば，x および y の値が正規分布乱数となっている．すなわち

$$\begin{aligned} p &= F^{-1}(r_1) = -\log(1 - r_1) \\ \theta &= 2\pi r_2 \\ x &= \sqrt{2p}\cos\theta = \sqrt{-2\log(1 - r_1)}\cos\theta \\ y &= \sqrt{2p}\sin\theta = \sqrt{-2\log(1 - r_1)}\sin\theta \end{aligned} \tag{11.14}$$

であり，乱数を 2 個使っているが，x と y の両方を使うことができるので，効率は落ちていない．図 11.5 はこの方法で作った正規分布乱数の分布である．

プログラム (Random_Normal.java) のメソッド (random_Normal) の一部と，その説明は以下のとおりである．

Random_Normal.java のメソッド (random_Normal) の一部：

```
 1  boolean sw= true;
 2  double p, th;
 3    public double random_Normal(double sigma) {
 4        if (sw) {
 5            sw= false;
 6            p= sigma*Math.sqrt(-2.0*Math.log(1.0-Math.random()));
 7            th= 2.0*Math.PI*Math.random();
 8            return p*Math.cos(th);
 9        } else {
10            sw= true;
11            return p*Math.sin(th);
12        }
13    }
```

1. L1, 2: 式 (11.14) の x と y を交互に求めるために，論理型変数 sw を使い，その初期値を true にしておく．変数 p, th は sw が false のときには，前の値を使うので，メソッドの中で定義せずに外でクラス内に出しておく．
2. L3: メソッド random_Normal の定義である．引数 sigma は正規分布の標準偏差であり，戻り値は正規分布乱数である．
3. L4: 論理変数 sw を調べる．
4. L5: sw の値が真であったら，次回のために sw を false にする．
5. L6, 7: 式 (11.14) の p, θ を計算する．
6. L8: 戻り値として式 (11.14) の x を返す．
7. L10: sw の値が偽であったら，次回のために sw を true にする．
8. L11: 戻り値として式 (11.14) の y を返す．
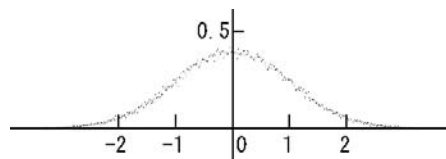
図 11.5　ボックス・マーラー法による正規分布乱数の分布．$\sigma = 1.0$ で x の刻みは 0.02，発生させた乱数の数は 10 万である．（Z11_05_siekibunpu.java）

11.2.4 球面一様分布乱数

半径 1 の球面上の座標は 3 次元極座標の θ, ϕ で表すことができる．一様に分布する乱数を座標 (θ, ϕ) の値で作る際に，ϕ については範囲を $(0, 2\pi)$ に拡張するだけでよいが，θ について一様にとると両極 $\theta = 0, \pi$ の付近で密になり，赤道 $\theta = \pi/2$ の付近で疎になってしまう．その比率は $\dfrac{1}{\sin\theta}$ である．しかし，$z = \cos\theta$ を一様にとれば，$|dz| = \sin\theta d\theta$ であるから球面上の θ 方向でも一様な分布となる．そこで，2 個の乱数 r_1, r_2 を作り，それから座標 (θ, ϕ) を作ると，

$$\begin{aligned} \theta &= \sin^{-1}(2(r_1 - 0.5)) \\ \phi &= 2\pi r_2 \end{aligned} \tag{11.15}$$

であり，座標 (x, y, z) を作ると

$$\begin{aligned} z &= 2(r_1 - 0.5) \\ y &= \sin\theta\sin\phi \\ x &= \sin\theta\cos\phi \end{aligned} \tag{11.16}$$

である．

図 11.6 は $4,000$ 組の座標を 3 次元の球面上にプロットしたものである．

プログラム (Random_Sphere.java) のメソッド (random_Sphere) と，その説明は以下のとおりである．

Random_Sphere.java のメソッド (random_Sphere)：

```
1   public double[] random_Sphere(double r) {
2     double[] ans= new double[3];
3     double pi2= 2*Math.PI;
4     double ph= Math.random()*pi2;
5     double z= (Math.random()-0.5)*2;
6     double rl= Math.sqrt(1.0-z*z);
7     ans[0]= r*rl*Math.cos(ph);
```
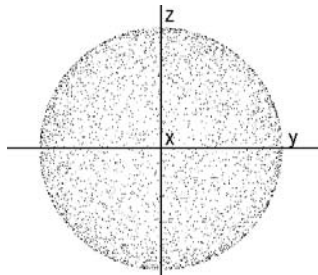
図 11.6　球面一様分布乱数の分布．3 次元球面上に一様に分布した点を投影した図である．(Z11_06_sphere.java)

```
 8     ans[1]= r*rl*Math.sin(ph);
 9     ans[2]= r*z;
10     return ans;
11   }
```

1. L1〜11: メソッド random_Sphere の定義である．引数 r は球の半径であり，戻り値は球面一様分布乱数（x, y, z）の配列である．
2. L2: 戻り値（x, y, z）の配列 ans の定義である．
3. L4: ϕ を算出する．
4. L5: θ の代わりに z を算出する．
5. L6: 赤道面上での動径を求める．
6. L7〜9: 戻り値（x, y, z）を配列 ans に代入する．
7. L10: 戻り値 ans を返す．

11.2.5　表読み取り法

これまでに述べた標準指数分布は単純な累積分布関数があるので，標準指数分布乱数を簡単な変換関数で作ることができる．正規分布乱数の場合は，単純な累積分布関数はないが，巧妙な細工をして作ることができた．また，球面一様分布乱数は変数として z, ϕ を使うことで作ることができた．しかし，このような細工ができない場合には，数値積分で累積分布関数の表を作り，r に対応する値を参照して x とする表読み取り法(table lookup)を使わなければならない．元来，乱数の研究には長い歴史があり，それは優れた一様分布を作るための努力であった．表読み取り法では，いくら表を細かくしても離散的であるから，この努力を無視することになってしまう．しかし，普通の場合には十分役立つであろう．

図 11.7 は任意に指定できる分布乱数の発生例である．ここでは 10 組の（x, y）のデータ値を与え，その累積分布関数をシンプソンの積分法で作り，乱数の発生をさせている．座標は [0, 500] の範囲を 100 分割し，乱数を 10 万回作っている．$\circ$ が指定したデータ値であり，$+$ は各座標値に分配された乱数の数である．縦軸は意味がないが，指定したデータの面積と，得られた乱数の分布の面積が同じ値になるように規格化してある．この分布は，後の仮想浅間降灰予測の図 11.9 の作成の際使われている．
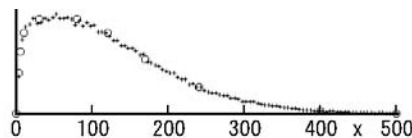
図 11.7　任意に指定できる分布乱数の分布．10 組の（x, y）のデータ値を与え，その分布を与える乱数の発生をさせる．座標は [0, 500] の範囲を 100 分割し，乱数を 10 万回発生させた．$\circ$ は指定したデータ値であり，$+$ は各座標値に分配された乱数の回数である．縦軸は意味がなく，両カーブに同一の規格化をしてある．（Z11_07_rand_table.java）

プログラム (Random_Function.java) のコンストラクタ (Random_Function) やメソッド (random_Function) とその説明は以下のとおりである．積分関数の逆関数がわかっている場合には，コンストラクタに引数はない．

Random_Function.java のメソッド (random_Function) の一つで，欲しい分布の型を表す関数の累積分布関数である原始関数 (積分) の逆関数がわかっている場合：

```
1  public double random_Function(MyFunction f) {
2    return f.function(Math.random());
3  }
```

1. L1〜3: メソッドの定義である．引数の f はインタフェース MyFunction クラスのインスタンスであり，欲しい分布の型を表す関数の累積分布関数の逆関数定義している．戻り値は座標値の乱数である．
2. L2: 累積分布関数の逆関数に乱数を与え，欲しい分布関数での座標値を求め，戻り値として返す．

累積分布関数の逆関数がわからない場合には，コンストラクタで初期処理をする必要がある．

Random_Function.java のコンストラクタ (Random_Function) の一つで，累積分布関数の逆関数がわからない場合には，累積分布関数の逆関数の表を作って，読み取るようにしなければならない．

Random_Function.java のコンストラクタ (Random_Function) の一つ：

```
 1  int np;
 2  double[] integ_f;
 3  public Random_Function(int np, MyFunction f) {
 4    this.np= np;
 5    integ_f= new double[np];
 6    double xd;
 7    double dx= 1.0/(np-1);
 8    double dx2= dx/2;
 9    double dx6= dx/6;
10    integ_f[0]= 0.0;
11    for (int i=1; i<np; i++) {
12      xd= (i-1)*dx;
13      integ_f[i]= integ_f[i-1]
14        +(f.function(xd)+4*f.function(xd+dx2)
15            +f.function(xd+dx));
16    }
17    double wnorm= 1.0/integ_f[np-1];
18    for (int i=1; i<np; i++) {
19      integ_f[i] *= wnorm;
20    }
21  }
```

1. L1, 2: 広域的変数 np と累積分布関数の配列 integ_f を宣言する．

2. L3〜21: コンストラクタの定義であり，引数の np は分布関数の座標の両端を含めた等分割点数であり，f はインタフェース MyFunction クラスのインスタンスであり，分布関数を定義している．
3. L4, 5: 引数の np の値を広域的変数 this.np に渡し，広域的配列 integ_f のインスタンスを作る．
4. L7: x 座標の刻み幅を求める．
5. L8, 9: シンプソン積分のための定数．
6. L10: 累積分布関数の最小値をゼロとする．
7. L11〜16: 累積分布関数作成のループである．
8. L12: 1 区間のシンプソン積分の始点の座標値．
9. L13〜15: 1 区間のシンプソン積分を行い，直前の配列の値に加えて，累積値とする．
10. L17: Math.random で作られる値の範囲が $[0, 1)$ であるから，最終値が 1 となるように規格化するための規格化因子を作る．
11. L18〜20: 規格化を行う．これで累積分布関数の表ができあがる．

Random_Function.java のメソッド (random_Function) の一つで，積分関数の逆関数がわからない場合：

```
 1     public double random_Function()  {
 2     double r= Math.random();
 3     double xd;
 4     double dx= 1.0/(np-1);
 5     int i;
 6     for (i=1; i<np-1; i++) {
 7       if (r<integ_f[i]) break;
 8     }
 9     xd= (i-1)*dx;
10     return (xd+dx*(r-integ_f[i-1])/(integ_f[i]-integ_f[i-1]));
11   }
```

1. L1〜11: メソッド random_Function () の定義であり，戻り値は座標が $[0, 1)$ の範囲に変換された乱数値である．
2. L2: 一様乱数の値を r に代入する．
3. L4: x 座標の刻み幅を決める．
4. L6〜8: 表の読み取りを行う．乱数値が累積値を超えたら，その乱数は超えた点以下の 1 区間にあることになる．
5. L9, 10: 座標値は乱数値が累積値を超えた点以下の 1 区間にあるから，線型逆内挿式で座標値を決定し，戻り値として返す．

プログラム (Z11_07_rand_table.java) のメインメソッドの一部と，その説明は以下のとおりである．

Z11_07_rand_table.java のメインメソッドの一部：

```
 1     int nx=101;
 2     double[] x= new double[nx];
 3     double[] y= new double[nx];
```

```
 4     double dx= (xmax-xmin)/(nx-1);
 5     for (int i=0; i<nx; i++) {
 6       x[i]= xmin+i*dx;
 7       y[i]= 0.0;
 8     }
 9     MyFunction mf= new Func();
10     Random_Function rf= new Random_Function(nx, mf);
11     int nmax= 100000;
12     int k;
13     for (int j=0; j<nmax; j++) {
14       k= (int)(rf.random_Function()*nx);
15       y[k]+= 1.0;
16     }
```

1. L1: 分布関数の範囲は，すでに xmin，xmax で [0.0, 100.0] に指定してあり，ここで刻み数を指定する．
2. L2, 3: 分布関数の刻みの x 座標と，戻される乱数値が当てはまる x 座標に累積される度数の配列 y の定義をする．
3. L4〜8: 座標 x と度数 y の初期化を行う．
4. L9, 10: MyFunction のインスタンスを作り，それを使って Random_Function のインスタンスを作る．
5. L11: 乱数を 10 万個作る．
6. L13〜16: 分布乱数を作り，当てはまる添字番号として y に 1 を累積する．

プログラム (Z11_07_rand_table.java) のクラス (Z11_07_rand_tableCommon) と (Func)，およびその説明は以下のとおりである．

Z11_07_rand_table.java のクラス (Z11_07_rand_tableCommon と Func)：

```
 1 class Z11_07_rand_tableCommon {
 2   double xmin, xmax;
 3 }
 4 class Func extends Z11_07_rand_tableCommon implements MyFunction {
 5   Lagrange la= new Lagrange();
 6   public double function(double x)  {
 7     double xw= xmin+x*(xmax-xmin);// [0,1]->[xmin,xmax]
 8     double[] xd= {0,2.5,  5,10,30,80,120,170,240,400,xmax};
 9     double[] yd= {0,  3,4.6, 6, 7, 7,  6,  4,  2, .2,   0};
10     return la.lagrange(xw,xd,yd,0,xd.length-1,3);
11   }
12 }
```

1. L1〜3: メインメソッドとクラス Func とで共通に使う変数 xmin，xmax をクラス変数として定義するクラス Z11_07_rand_tableCommon を作る．
2. L4〜12: 分布関数を与える Func クラスの定義．クラス Z11_07_rand_tableCommon を継承し，インタフェース MyFunction を実装する．
3. L5: ラグランジュの内挿法を使うために，インスタンスを作る．

4. L6〜11: インタフェース MyFunction の抽象メソッド function をオーバーライドして，関数を定義している．
5. L7: 引数 x は $[0, 1)$ で与えられるから，実際の範囲 [xmin, xmax) に変換する．
6. L8〜10: 関数をデータで与えているので，ラグランジュの内挿法で関数値を求めている．

11.3　シミュレーション

11.3.1　ビュフォンの針

図 11.8 はビュフォンの針の問題である．左図は紙の上に間隔 1 の線を多数引き，長さ $L \le 1$ の多数の針を無作為に落としたときの図であり，針がどれかの線と交わる確率は $2L/\pi$ になる．ここで行ったシミュレーションでは，間隔が 1 の線を 5 本引き，針の中心を x 軸についても y 軸についても $(-2, 2)$ の範囲にランダムに落とすようにした．針の向きもランダムである．

確率を求める数学は，以下のとおりである．y 軸方向については y 依存性はないから，考慮する必要はない．x 軸方向については周期 1 の周期性と，各線に関する左右の対称性，針の向きの上下の対称性を考慮すると，x 軸に関して針の中心が $(-\frac{1}{2}, 0)$ に落ちて，針の向きが右上を向いている場合で考慮すればよいことになる．そこで，長さ L の針が線上に落ちる確率は，右図のような作図をすると

$$
\begin{aligned}
P(L) &= 2\left\{\int_0^{L/2} \frac{\theta}{\pi/2} dx + \int_{L/2}^{1/2} 0\, \mathrm{d}x\right\} \\
&= -\frac{2L}{\pi}\int_{\frac{\pi}{2}}^{0} \theta \sin\theta \mathrm{d}\theta = \frac{2L}{\pi}\int_0^{\frac{\pi}{2}} \theta \sin\theta \mathrm{d}\theta = \frac{2L}{\pi}
\end{aligned}
\tag{11.17}
$$

となる．ここでは，$\cos\theta = \dfrac{x}{L/2}$，$dx = -\dfrac{L}{2}\sin\theta \mathrm{d}\theta$ と $\displaystyle\int_0^{\frac{\pi}{2}} \theta \sin\theta \mathrm{d}\theta = 1$ を使った．右
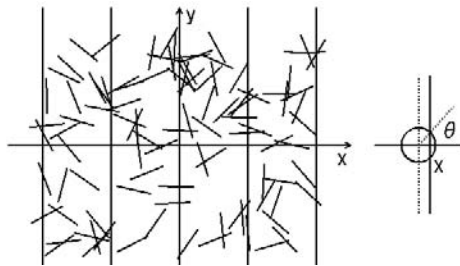
図 11.8　ビュフォン (Buffon) の針．左図は紙の上に間隔 1 の線を多数引き，長さ $L \le 1$ の多数の針を無作為に落とすと，針がどれかの線と交わる確率は $2L/\pi$ になる．右図はこの確率を数式で求めるときの補助の図である．（Z11_08_buffon.java）

図の円は，針の中心が円の中心に落ちたときの針が存在し得る領域である．x は円の中心と実線との距離であり，θ は x 軸から上向きにとった円と直線との交点までの角である．このとき，針が実線に乗る確率は $\dfrac{\theta}{\pi/2}$ である．x 軸での積分をする際には，実際は線は動かず円が移動するはずであるが，円は動かず線がゼロから 1/2 まで移動すると考えたほうがわかりやすいであろう．

プログラム (Z11_08_buffon.java) のメインメソッドの一部と，その説明は以下のとおりである．

Z11_08_buffon.java のメインメソッドの一部：

```
 1     double Length= 0.5;
 2     double L2= Length/2;
 3     int in=0;
 4     double xl= (xmax-xmin-1);
 5     double xl2= xl/2;
 6     double yl= (ymax-ymin-1);
 7     double yl2= yl/2;
 8     double theta;
 9     double pi2= Math.PI/2;
10     double p1x, p2x, p1y, p2y;
11     double costh, sinth;
12     for (j=0; j<max; j++) {
13       x= xl*Math.random()-xl2;
14       y= yl*Math.random()-yl2;
15       theta= Math.PI*Math.random()-pi2;
16       costh= Math.cos(theta);
17       sinth= Math.sin(theta);
18       p1x= x+L2*costh;
19       p2x= x-L2*costh;
20       p1y= y+L2*sinth;
21       p2y= y-L2*sinth;
22       if (j<100) gk.gjoin(p1x, p1y, p2x,p2y);
23       for (i=-2; i<=2; i++) {
24         x= i;
25         if (p2x<=x && x<=p1x) {
26           in++;
27           break;
28         }
29       }
30     }
```

1. L1: 縦線の間隔を 1 としてあるので，針の長さを間隔の半分にするため 0.5 を指定する．
2. L2: 針の長さの半分．
3. L3: ヒットした数を数えるカウンタ変数をゼロで初期化する．
4. L4, 5: 針の中心が落ちる x 軸上の幅 xl とその半分の xl2.
5. L6, 7: 針の中心が落ちる y 軸上の幅 yl とその半分の yl2.
6. L12〜30: 針を落とす回数だけまわすループ．

7. L13, 14: 針の中心が落ちる x 座標と y 座標.
8. L15: 針の向きの角度.
9. L18〜21: 針の先端 p1 と末端 p2 の座標の計算.
10. L22: 便宜上 100 回以下なら針を表示する.
11. L23〜29: 線の数だけループをまわし，針が線のどれかに触れるかどうかを調べる.
12. L24: 線の位置を x に代入する.
13. L25〜28: 針の末端が線より左にあり，先端が線より右にあれば，針は線に触れているから，カウンタに 1 を加え，ループを抜けて針を落とすループを繰り返す.
14. L30: 針を落とすループの終了.

11.3.2　仮想浅間降灰予測

図 11.9 は風速 $4\,\mathrm{ms}^{-1}$ (左図) と $8\,\mathrm{ms}^{-1}$ (左図) の北西の風による，仮想浅間の降灰の分布予測図である．同心円の刻み幅は 20 km である．与えるパラメータは，風上の向きと風速である．原点からの距離に対する降灰量は，図 11.7 に示してある分布乱数を，風速 $5\,\mathrm{ms}^{-1}$ のときの分布とし，風速 $v\,\mathrm{ms}^{-1}$ の場合は，距離の値を $\sqrt{\dfrac{v}{5}}$ 倍することにしている．すべてのデータや関数は仮想のものであり，現実の理論やデータを利用しているわけではない．

風速の弱い場合は，分布が広がる角度が大きくなるようにしてある．その説明は以下のとおりである．噴煙は原点で等方的に起こり，その後，軽い粒子ほど風に乗って遠距離まで飛んでいく．そこで，原点を放物線の焦点とし，その放物線の内側に多く分布するとする．その角度分布は，正規分布とし，放物線の中心線を分布の中心，両側の放物線を正規分布の $\pm 2\sigma$ として，正規分布型の乱数を使う．

焦点が原点 $(0,0)$ に，頂点が $(0,\,-c)$ にある放物線の方程式は

$$y = \frac{1}{4c}x^2 - c \tag{11.18}$$

であり，その図は図 11.10 である．半径 r の円との交点は,
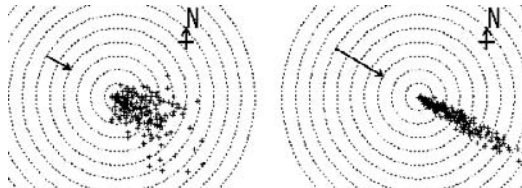
図 11.9　仮想浅間降灰予測．風上の向きと風速を指定して降灰分布の予測を行う．風上は両図とも北から西へ 60° であり，風速は左図では $4\,\mathrm{ms}^{-1}$，右図では $8\,\mathrm{ms}^{-1}$ である．同心円の刻み幅は 20 km である．
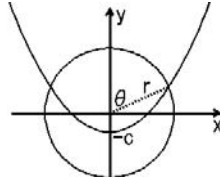
図 11.10　焦点を原点に置く放物線と円の交点の関係.

$$
\begin{aligned}
y &= \frac{1}{4c}x^2 - c \\
x^2 + y^2 &= r^2
\end{aligned} \tag{11.19}
$$

の解（$\pm\sqrt{4c(r-c)},\ r-2c$），ただし $r \geq c$ であり，中心線からの角度は,

$$
\theta = \cos^{-1}\left(1 - \frac{2c}{r}\right) \tag{11.20}
$$

となる．速度 v と c の関係は適当であり

$$
c = 10^{-\frac{1}{3}(v-10)-1} \tag{11.21}
$$

としている.

プログラム (Z11_09_MtAsama.java) のメインメソッドの一部と，その説明は以下のとおりである.

Z11_09_MtAsama.java のメインメソッドの一部：

```
 1     double v= 4;
 2     double scale= Math.sqrt(v/5);
 3     double wind= -60;
 4     double wind_rad= wind/180*Math.PI;
 5     double r, c, theta;
 6   中略
 7     double x,y, sinw, cosw;
 8     int i;
 9   中略
10     int npoint=201;
11     MyFunction mf= new Func();
12     Random_Function rf= new Random_Function(npoint, mf);
13     c= Math.pow(10.0, -(v-10)/3-1);
14     for (i=1; i<ndot; i++) {
15       r= rf.random_Function() * npoint * scale;
16       if (r<c) {
17         theta= 180.0;
18       } else {
19         theta= Math.acos(1-2*c/r);
20       }
21       Random_Normal rn= new Random_Normal();
22       theta= rn.random_Normal(theta/2);
```

```
23        x= r*Math.sin(theta-wind_rad);
24        y= -r*Math.cos(theta-wind_rad);
25        gk.gmark(x, y);
26      }
```

1. L1〜5: 風速，風向き等の指定をする．
2. L10: 動径座標の刻み数を決める．
3. L11: Func 型インスタンスの作成をする．
4. L12: 座標の刻み数と Func 型インスタンスを引数として Random_Function 型のインスタンスを作成する．
5. L13: 式 (11.21) による放物線の頂点の位置を決める．
6. L14〜26: 指定した点の数 ndot だけループをまわす．
7. L15: 指定した分布関数による乱数を使って，動径座標を決める．
8. L16〜20: 動径座標に依存する角度分布の範囲を指定する．
9. L21, 22: 正規分布乱数を使って，角度座標を決定する．
10. L23〜25: x, y 座標を求め，点を描画する．

演習問題

11.1　乱数を使ったヒット・オア・ミス法のプログラム Z11_01_monte.java を変更して，じゃんけんのプログラム Q11_01_Z11_01_janken.java を作りなさい．二人の名前は a さんと b さんで，乱数の値が $[0, g]$ なら「ぐう」，$[g, c)$ なら「ちょき」，$[c, 1)$ なら「ぱー」とするように，前もってソースプログラムで配列に a さんと b さんの g と c の値を代入しておきなさい．a さんを横軸，b さんを縦軸にとり，1 回のじゃんけんごとに乱数値を座標値としてマークをプロットし，その色は a さんが勝ったら赤，b さんが勝ったら青とし，「あいこ」の場合は黄色としなさい．じゃんけんの回数 n は適当に指定すること．

解答例： ソースプログラム Q11_01_Z11_01_janken.java

11.2　前問 11.1 のプログラムを変更して，Q11_02_Z11_01_janken.java を作り，キーボードから a さんと b さんの g と c の値を，「何割」の単位で入力するようにしなさい．このプログラムは，入出力があるから html 文書としては使えない．

解答例： ソースプログラム Q11_02_Z11_01_janken.java

第 12 章

スプライン関数と作図

これまで扱った補間法および関数近似と平滑法において，スプライン関数を使う方法も取り上げたが，ここではスプライン関数の基礎について解説し，作図への応用例を述べる．計算機で図を描けるようになった 1970 年代半ばまで，図は主に手書きで行われていた．紙の上に曲線を描く場合には，適当な間隔で点をプロットし，順次，並んだ 4 点が乗るような雲形定規[1]を選んだり，あるいは，4 点を通るように滑らかに曲げることのできる自在定規を使って，4 点の内部の 2 点間のみで線を描く作業を繰り返すことで行われた．スプラインとは自在定規を意味する言葉であり，実際の自在定規を使って描かれる曲線は，3 次のスプライン関数に相当している．

広い領域の多くのデータ点を使って，関数近似や補間をする場合に，ラグランジュの方法などでは，1 個の多項式を使うと高い次数が必要となり，余計な振動が入って精度が良くないことがある．使用するデータ点の数を少なくして，局所的にラグランジュの方法を適用すれば関数値としては精度を上げることができるが，広い領域にわたって見ると，両側で異なった多項式を使うデータ点においては，左右の微分値が一致するとは限らない．一方，スプライン関数では，データ点 (正確にいうと節点) においても n 次式の場合には $n-1$ 階微分まで可能となっている．しかし，スプライン関数の次数を上げると，処理時間が大幅に増えるので 3 次のスプライン関数が手頃であり，それが多く使われている．

スプライン関数の要点を一言でいうなら，それは切断べき関数にあるといえよう．まず，切断べき関数について述べる．

12.1 切断べき関数

$x=\xi$ を節点とする n 次の切断べき関数の定義は

$$(x-\xi)_+^n = \begin{cases} (x-\xi)^n & (x \geq \xi) \\ 0 & (x \leq \xi) \end{cases} \tag{12.1}$$

1. 多様な曲線から作られた定規のセットがあった．

である．図 12.1 は節点 $\xi = 0$ における 3 次の切断べき関数である．

いま，$x = \xi$ の両側で異なる n 次式で表される関数があったとすると，それは

$$S_n(x) = P_n(x) + c(x - \xi)_+^n \tag{12.2}$$

で表すことができる．ここで，$P_n(x)$ は $x \leq \xi$ で成り立つ n 次式であり，c を適当に決めれば，$P_n(x) + c(x - \xi)_+^n$ は $x \geq \xi$ で成り立つ n 次式とすることができるからである．節点 ξ_i が $i = 1$ 〜 ν の ν 個ある場合に，このことを繰り返すと，スプライン関数

$$S_n(x) = P_n(x) + \sum_{i=1}^{\nu} c_i(x - \xi_i)_+^n \tag{12.3}$$

を作ることができる．ここで，

$$S_{n,\,j}(x) = P_n(x) + \sum_{i=1}^{j} c_i(x - \xi_i)_+^n \tag{12.4}$$

と置くと，

$$S_{n,\,j}(x) - S_{n,\,j-1}(x) = c_j(x - \xi_j)_+^n \tag{12.5}$$

となる．この式 (12.5) を n 階微分すると

$$S_{n,\,j}^{(n)}(x) - S_{n,\,j-1}^{(n)}(x) = c_j n! \tag{12.6}$$

となるから，$x = \xi_j$ と置くことによって

$$\begin{aligned} c_j &= \frac{1}{n!}\left\{S_{n,\,j}^{(n)}(\xi_j) - S_{n,\,j-1}^{(n)}(\xi_j)\right\} \\ &= \frac{1}{n!}\left\{S_{n,\,j}^{(n)}(\xi_j) - S_{n,\,j-1}^{(n)}(\xi_j)\right\} \end{aligned} \tag{12.7}$$

となる．これで c_i は一意的に決まることがわかった．c_i を具体的に求めるには式 (12.3) に節点での値を代入して連立方程式を解けばよいが，これはあまり大きくない ν の場合でも条件の良くない (ill-conditioned) 連立方程式であり，計算の過程で桁落ちの誤差が大きく入ってくるので，このままでは実用的ではない．そこで，いろいろな工夫がなされており，次節ではその一つである B スプラインについて述べる．
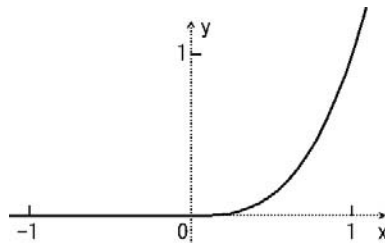
図 12.1　節点 $\xi = 0$ における 3 次の切断べき関数．$x \leq 0$ で $y = 0$，$x \geq 0$ で $y = x^3$ であり，$x = 0$ でも 2 階微分までは連続である．

12.2　B スプライン

スプライン関数の作り方にはいくつかの方法があるが，ここではよく使われる B スプラインを取り上げる．B スプラインの英語は basis spline であり，局所的な基底関数である．m 階の B スプラインは，$m+1$ 個の節点を使い，$m-1$ 次の多項式である．ここでは，4 階 (3 次) のスプライン関数を取り上げることにする．その定義は以下のとおりである．まず，各 ξ_j における切断べき関数

$$M_4^{1,j}(x) = (\xi_j - x)_+^3 = \begin{cases} (\xi_j - x)^3 & (x \le \xi_j) \\ 0 & (x \ge \xi_j) \end{cases} \tag{12.8}$$

を作る．$M_4^{1,j}(x)$ の 4 は階数 m であり，1 は使用する節点の数，j は使用する節点のうち最も左にある節点 ξ_j の番号である．4 階のスプライン関数は，切断べき関数の 4 階の差分商であり，

$$\begin{aligned} M_4^{2,j}(x) &= \frac{M_4^{1,j+1}(x) - M_4^{1,j}(x)}{\xi_{j+1} - \xi_j}, \quad M_4^{3,j}(x) = \frac{M_4^{2,j+2}(x) - M_4^{2,j}(x)}{\xi_{j+2} - \xi_j} \\ M_4^{4,j}(x) &= \frac{M_4^{3,j+3}(x) - M_4^{3,j}(x)}{\xi_{j+3} - \xi_j}, \quad M_4^{5,j}(x) = \frac{M_4^{4,j+4}(x) - M_4^{4,j}(x)}{\xi_{j+4} - \xi_j} \end{aligned} \tag{12.9}$$

で求められる最後の $M_4^{5,j}(x)$ である．

図 12.2 は節点が ξ_0 から ξ_6 までで，重複節点がない場合の $M_4^{i,j}(x)$ である．最も上のグラフは ξ_0 から ξ_6 までの 7 個の 3 次切断べき関数 $M_4^{1,j}(x)$ である．最も下のグラフは ξ_0 から ξ_2 までの 3 個の基底関数 $M_4^{5,j}(x)$ である．上から下へ節点の数が増すに従って関数の数は 1 個ずつ減っている．この基底関数は 2 次導関数まで連続であり，一般的に m 階では $m-2$ 次導関数まで連続である．

図 12.3 は重複節点がある場合であり，節点の x 座標は 0, 1, 1, 2, 3, 3, 3, 4, 5, 5, 5, 5 である．最も上のグラフは 3 次式が使われている切断べき関数が，ξ_0 から ξ_{11} まで 12
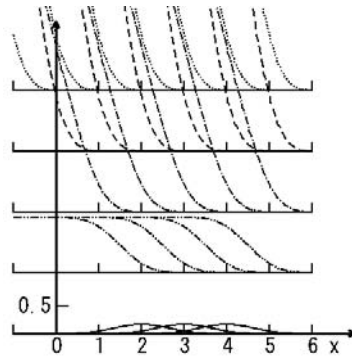
図 12.2　4 階 (3 次) の B スプライン．上から i 番目が $M_4^{i,j}(x)$ である．$M_4^{i,j}(x)$ は ξ_j から ξ_{j+i-1} までの i 点における切断べき関数を使ってできる差分商であり，$M_4^{5,j}(x)$ は ξ_j から ξ_{j+4} の 5 点を使う基底関数である．
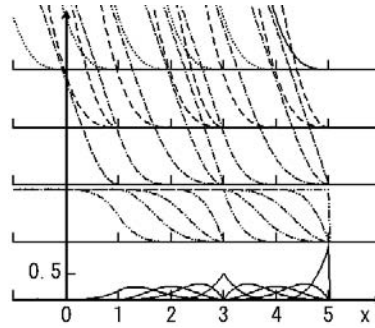
図 12.3　重複節点のある 4 階 (3 次) の B スプライン．節点の x 座標は 0, 1, 1, 2, 3, 3, 3, 4, 5, 5, 5, 5 である．一番上は 3 次の切断べき関数であり，一番下が基底関数 $M_4^{5,j}(x)$ である．

個あるが，$x=1$ で 2 重，$x=3$ で 3 重，$x=5$ で 4 重となっている．最も下のグラフには 8 個の基底関数がある．単節点の ξ_0 から始まる関数は，節点 0, 1, 1, 2, 3 を使っているから，1 で 2 次導関数は不連続である．それ以外の節点では 2 次導関数まで連続である．2 重節点の ξ_1 から始まる 2 個の関数のうち，左側の関数は節点が 1, 1, 2, 3, 3 であり，1, 3 で 2 次導関数は不連続である．右側の関数は節点が 1, 2, 3, 3, 3 であるから，3 では 1 次導関数も不連続である．左から 4,5 番目の関数も 3 では 1 次導関数が不連続である．8 番目の関数の節点は 4, 5, 5, 5, 5 であるから，5 が 4 重節点であり，関数値も不連続になっている．このように，節点を重複することによって，いろいろな不連続性をもつ基底関数を作ることができる．

このように，m 階 $(m-1$ 次$)$ の B スプラインは，m 階の切断べき関数を元に作ることも可能であり，関数の連続性は切断べき関数の性質を受け継ぎ，重複節点でなければ，$m-2$ 次導関数まで連続である．この方法で作られる B スプラインは，以前のドュブア・コックスのアルゴリズムによる B スプラインと同じものであるが，B スプラインの性格がよくわかるので，ここでの解説で取り上げた．これまでに使った規格化された B スプラインとは式 (3.29) の関係

$$N_{m,j}(x) = (\xi_j - \xi_{j-m}) M_m^{m+1,j}(x) \tag{12.10}$$

がある．

12.3　リーゼンフェルトの作図

リーゼンフェルトの方法によるスプライン関数を使うと，2 次元曲線を作成することができる．2 次元開曲線と 2 次元閉曲線では，データ点のとり方が異なるので，それぞれを別々に取り扱うことにする．

12.3.1　2 次元開曲線

これまで B スプラインを使って補間，平滑化などについて述べたが，ここでは平面 2 次元曲線の生成に使うリーゼンフェルトの方法について述べる．$y = f(x)$ の関係があるデータの補間あるいは平滑化の場合には，B スプラインを基底関数とした線形結合でスプライン関数

$$S(x) = \sum_{j=1}^{n} c_j N_{m,j}(x) \tag{12.11}$$

を定義し，補間の場合には，このスプライン関数がデータ点 (x_i, y_i) の上を通るという条件を使って係数 c_i を決定した．平滑化の場合には，B スプラインの数より多いデータ点を使って，最小二乗法により係数 c_i を決定した．また，(x, y) 平面上の曲線の場合には，媒介変数 t を使って $x = x(t)$, $y = y(t)$ のように二つの関数とし，それぞれに式 (12.11) を適用して

$$\begin{aligned} x(t) &= \sum_{j=1}^{n} c_j^x N_{m,j}(t) \\ y(t) &= \sum_{j=1}^{n} c_j^y N_{m,j}(t) \end{aligned} \tag{12.12}$$

とした．そして，この式にデータ点の値を代入して連立方程式を作り，係数 c_j^x と c_j^y を決定して，座標値 $(x(t), y(t))$ を求めた．そこで，補間の場合には曲線はデータ点の上を通ることになっている．

一方，リーゼンフェルトの方法では，スプライン関数を

$$\begin{aligned} x(t) &= \sum_{j=1}^{n} x(t_j) N_{m,j}(t) \\ y(t) &= \sum_{j=1}^{n} y(t_j) N_{m,j}(t) \end{aligned} \tag{12.13}$$

のように定義して，座標値 $(x(t), y(t))$ を求める．t_j はデータ点であり，$(x(t), y(t))$ はデータ値である．この方法によると，曲線は一般的にデータ点の上を通らず，前後のデータのある平均値のような値をとることになる．そこで，データ点の上を通るようにしたい場合には，そのデータ点自体を重複させることになる．

式 (12.13) の和の項の中でゼロでない $N_{m,j}(t)$ は，t の値の周囲 m 点 (m は階数) の t_j に対応する関数 $N_{m,j}(t)$ だけであるから，1 点のデータ値に変化が生じても，その影響は周りの数点に及ぶだけであり，極めて局所化している．そこで，図形の作成に適しており，CAD などに使われている．

図 12.4 は，リーゼンフェルトの方法による 2 次元開曲線の例である．図形の基本データの数 n は 8 点 (∘ 印) であり，表 12.1 に与えられている．補間法の場合と同様に，$n = \nu + m$ が満たされなければならない．ν は内部節点の個数，m は B スプラインの階数である．ここでは，3 次のスプラインを使うから $m = 4$ であり，$\nu = 4$ となる．
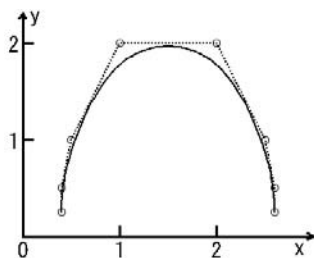
図 12.4　B スプラインによる 2 次元開曲線図形の作成．○ 印がデータ点であり，実線が作図開曲線である．

表 12.1　リーゼンフェルトの方法による 2 次元開曲線図形のデータ点．8 組あり，図 12.4 の上では ○ 印で示されている．

j	0	1	2	3	4	5	6	7
x_j	0.4	0.4	0.5	1.0	2.0	2.5	2.6	2.6
y_j	0.25	0.5	1.0	2.0	2.0	1.0	0.5	0.25

ここでも，媒介変数を t として，t 座標での節点を作る．それは，両端の付加節点を加えて

$$\xi_t = 0.0,\ 0.0,\ 0.0,\ 0.0,\ 1.0,\ 2.0,\ 3.0,\ 4.0,\ 5.0,\ 5.0,\ 5.0,\ 5.0 \tag{12.14}$$

の 12 点である．

プログラム (Bspl_Open.java) のメソッド (bspl_Open) と，その説明は以下のとおりである．

Bspl_Open.java のメソッド (bspl_Open)：

```
 1  public void bspl_Open(double[] xd,double[] yd,
 2      double[] xp, double[] yp) {
 3    int nd= xd.length;
 4    int np= xp.length;
 5    double tw, dt, x, y;
 6    int i, j, irj, ibspl;
 7    int ndp4= nd+4;
 8    int ndm4= nd-4;
 9    int ndm3= nd-3;
10    int ndm2= nd-2;
11    double[] t= new double[ndm2];
12    double[] xi= new double[ndp4];
13    double[] N= new double[4];
14
15    Bspl bs= new Bspl();
16
```

```
17      for (i=0; i<4; i++) {
18        xi[i]= 0.0;
19      }
20      for (i=0; i<ndm4; i++) {
21        xi[i+4]= i+1;
22      }
23      for (i=0; i<4; i++) {
24        xi[nd+i]= ndm3;
25      }
26
27      for (i=0; i<ndm2; i++) {
28        t[i]= i;
29      }
30
31      dt= (t[ndm3]-t[0])/(np-1);
32      for (i=0; i<np; i++) {
33        tw=t[0]+dt*i;
34        ibspl= bs.bspl(tw, xi, N);
35        x=0.0;
36        y=0.0;
37        for (j=0; j<4; j++) {
38          irj=ibspl+j-4;
39          x=x+xd[irj]*N[j];
40          y=y+yd[irj]*N[j];
41        }
42        xp[i]=x;
43        yp[i]=y;
44      }
45      return;
46    }
```

1. L1, 2〜46: メソッド bspl_Open の定義である．引数の xd, yd は開曲線を求める際に基本とするデータ点の x, y 座標の配列である．xp, yp は戻される開曲線の座標の配列であり，適当な大きさの配列としておく必要がある．
2. L3, 4: nd はデータ点の個数，np は求める曲線の配列の大きさである．
3. L7: 3 次の B スプラインを使うので，節点の数はデータ点より 4 点多くする必要がある．
4. L11: t は媒介変数であり，データ点の数より 2 点少ない配列である．
5. L12: xi は節点の配列である．
6. L13: N は B スプライン関数値の配列である．
7. L15: Bspl のインスタンスを作る．
8. L17〜25: 節点の最初の 4 点の値をゼロとし，その後は 1 ずつ増やした値として，最後の 4 点は同じ値とする．そこで，曲線は両端の点には接線として入るようになる．
9. L27〜29: 媒介変数にゼロから整数を順に代入する．
10. L31: 解となる変数 xp, yp と媒介変数の刻みを合わせる．
11. L32〜44: xp, yp の点の数だけループをまわす．
12. L33: 媒介変数の値を計算する．

13. L34: B スプラインの計算結果を配列 N に求め，データ配列での B スプラインの位置を ibspl に代入する．
14. L35〜43: 式 (12.13) で x，y のスプライン関数値を求め，配列 xp，yp に代入する．
15. L44: xp，yp のループを終える．

このように，実際にはこれまでに使った B スプラインのクラスを使っているが，どのようなからくりで作図の数値が決定されるかを調べるために，少し異なった見方をする．ここで，式 (12.13) に代わる式を x について書きなおすと

$$x_j^{[k]}(t) = \begin{cases} x_j & (k = 0 \text{ のとき}) \\ \lambda x_j^{[k-1]}(t) + (1-\lambda) x_{j-1}^{[k-1]}(t) & (k > 0 \text{ のとき}) \end{cases} \tag{12.15}$$

である．ここで，

$$\lambda = \frac{t - t_{j+1-m}}{t_{j+1-k} - t_{j+1-m}} \tag{12.16}$$

である．y についても同様である．

いま，$t = 2.25$ として具体的な計算をする．表 12.1 より $x_j^{[0]}(2.25) = x_j$ $(j = 2 \sim 5)$ であり，既知である．以下，順に $k = 1$ は

$$\left.\begin{aligned} &x_3^{[1]}(2.25) = \lambda x_3^{[0]}(2.25) + (1-\lambda) x_2^{[0]}(2.25) = 0.875 \\ &\lambda = \frac{2.25 - 0}{3 - 0} = 0.75 \end{aligned}\right\}$$
$$\left.\begin{aligned} &x_4^{[1]}(2.25) = \lambda x_4^{[0]}(2.25) + (1-\lambda) x_3^{[0]}(2.25) = 1.417 \\ &\lambda = \frac{2.25 - 1}{4 - 1} = 0.417 \end{aligned}\right\} \tag{12.17}$$
$$\left.\begin{aligned} &x_5^{[1]}(2.25) = \lambda x_5^{[0]}(2.25) + (1-\lambda) x_4^{[0]}(2.25) = 2.042 \\ &\lambda = \frac{2.25 - 2}{5 - 2} = 0.083 \end{aligned}\right\}$$

$k = 2$ は

$$\left.\begin{aligned} &x_4^{[2]}(2.25) = \lambda x_4^{[1]}(2.25) + (1-\lambda) x_3^{[1]}(2.25) = 1.214 \\ &\lambda = \frac{2.25 - 1}{3 - 1} = 0.625 \end{aligned}\right\}$$
$$\left.\begin{aligned} &x_5^{[2]}(2.25) = \lambda x_5^{[1]}(2.25) + (1-\lambda) x_4^{[1]}(2.25) = 1.495 \\ &\lambda = \frac{2.25 - 2}{4 - 2} = 0.125 \end{aligned}\right\} \tag{12.18}$$

$k = 3$ は

$$\left.\begin{aligned} &x_5^{[3]}(2.25) = \lambda x_5^{[2]}(2.25) + (1-\lambda) x_4^{[2]}(2.25) = 1.284 \\ &\lambda = \frac{2.25 - 2}{3 - 2} = 0.25 \end{aligned}\right\} \tag{12.19}$$

となり，$x(2.25) = 1.284$ が決定される．これらの式で計算される点は図 12.5 に ∘ 印で示されている．それぞれが，1 階前の 2 点を結ぶ線上の内分点であり，式 (12.17) と式 (12.18) の 2 点を結ぶ 3 本の線が破線で示されている．

12.3.2　重複データ点

リーゼンフェルトの方法によるスプライン曲線の作成で，特異点を作るには同じデータを複数並べる重複データ点を使うことになる．図 12.6 は重複データ点の効果を示す図である．一番下の曲線のデータ点には重複はなく，作図曲線は滑らかで，曲がり角のデータ点からずれている．下から 2 番目は，すべての曲がり角のデータ点は 2 重としてあり，下から 3 番目は 3 重としてある．
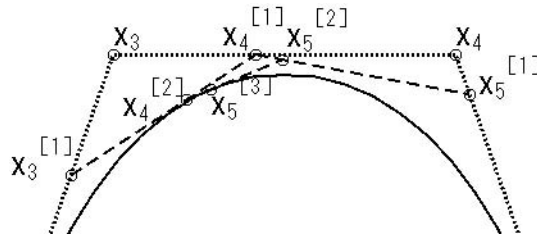
図 12.5　リーゼンフェルトの方法によるスプライン曲線上の点の算法．∘ 印は式 (12.17)〜式 (12.19) で計算される点であり，実線が作図開曲線である．図 12.4 の上部が拡大されている．
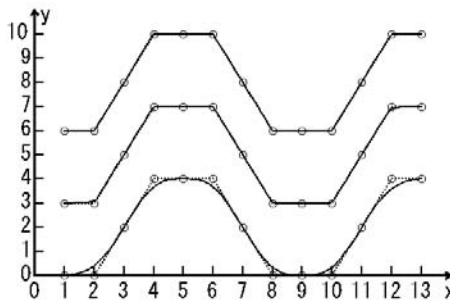
図 12.6　B スプラインによる 2 次元図形上の特異点の作成．∘ 印はデータ点であり，点線はデータ点を結ぶ直線，実線は作図曲線である．一番下の曲線のデータ点には重複はなく，下から 2 番目はすべての曲がり角のデータ点は 2 重としてあり，下から 3 番目は 3 重としてある．(Z12_06_Bspl_Riesenfeld_Mult)

12.3.3　2 次元閉曲線

2 次元閉曲線の場合は，配列の最初の要素と最後の要素に同じデータを与えておく必要がある．図 12.7 は ♡ 型の作図である．閉曲線であるから，どこから始めても同じであるが，ここでは，(1.25, 0.35) の ⊗ 印から始められている．$x = 0$ での上下二つの尖点は，3 重データ点を与えて作られている．

表 12.2 は ♡ 型閉曲線図形のデータ点である．

プログラム (Bspl_Closed.java) のメソッド (bspl_Closed) と，その説明は以下のとおりである．

Bspl_Closed.java のメソッド (bspl_Closed)：

```
1   public void bspl_Closed(double[] xd, double[] yd,
2       double[] xp, double[] yp) {
3     int nd= xd.length;
4     int np= xp.length;
5     double tw, dt, x, y;
6     int i, j, irj, ibspl;
```
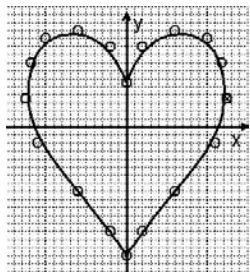
図 12.7　B スプラインによる 2 次元 ♡ 型曲線図形の作成．○ 印がデータ点であり，実線が作図曲線である．データ点は ⊗ 印から始まり，左回りにまわって，再び，始点と同じ点で閉じている．

表 12.2　♡ 型曲線図形のデータ点．データ点は (1.25, 0.35) の ⊗ 印から始まり，左回りにまわって，(0.0, 0.55) の 3 重点を通り，左半分を下りて (0.0, −1.60) の 3 重点を通り，再び，始点と同じ (1.25, 0.35) 点で閉じる．

x	1.25	1.18	1.00	0.60	0.20	0.00	0.00	0.00
y	0.35	0.80	1.10	1.20	1.00	0.55	0.55	0.55
x	−0.20	−0.60	−1.00	−1.18	−1.25	−1.10	−0.60	−0.20
y	1.00	1.20	1.10	0.80	0.35	−0.20	−0.80	−1.30
x	0.00	0.00	0.00	0.20	0.60	1.10	1.25	
y	−1.60	−1.60	−1.60	−1.30	−0.80	−0.20	0.35	

```
 7      int ndp6= nd+6;
 8      double[] t= new double[nd];    //  媒介変数
 9      double[] xi= new double[ndp6]; //  媒介変数用節点
10      double[] N= new double[4];     //  B スプライン
11
12      Bspl bs= new Bspl();
13
14      for (i=0; i<ndp6; i++) {
15        xi[i]= i-3;
16      }
17      for (i=0; i<nd; i++) {
18        t[i]= i;
19      }
20      dt= (t[nd-1]-t[0])/(np-1);
21      for (i=0; i<np; i++) {
22        tw=t[0]+dt*i;
23        ibspl= bs.bspl(tw, xi, N);
24        x=0.0;
25        y=0.0;
26        for (j=0; j<4; j++) {
27          irj=ibspl+j-4;
28          if(irj > nd-1) irj= irj-nd+1;
29          x=x+xd[irj]*N[j];
30          y=y+yd[irj]*N[j];
31          }
32        xp[i]=x;
33        yp[i]=y;
34      }
35      return;
36    }
```

1. L1, 2〜36: メソッド bspl_Closed の定義である．引数の xd, yd は閉曲線を求める際に基本とするデータ点の x, y 座標の配列である．xp, yp は戻される閉曲線の座標の配列であり，適当な大きさの配列としておく必要がある．
2. L3, 4: nd はデータ点の個数，np は曲線の配列の大きさである．
3. L7: 3 次の B スプラインを使うので，節点の数はデータ点より 6 点多くする必要がある．
4. L8: t は媒介変数であり，データ点と数が同じ配列である．
5. L9: xi は節点の配列である．
6. L10: B スプライン関数値の配列である．
7. L12: Bspl のインスタンスを作る．
8. L14〜16: 節点の値を指定するが，4 番目の要素の値がゼロになるように 3 ずらした値とする．
9. L17〜19: 媒介変数にゼロから整数を順に代入する．
10. L20: 解となる変数 xp, yp と媒介変数の刻みを合わせる．
11. L21〜34: xp, yp の点の数だけループをまわす．
12. L22: 媒介変数の値を指定する．
13. L23: B スプラインの計算結果を配列 N に求め，データ配列での B スプライン

の位置を ibspl に代入する．

14. L26〜33: 式 (12.13) で x，y のスプライン関数値を求め，配列 xp，yp に代入する．
15. L28: 使用するデータの添字番号が最大値を超えたら，(nd−1) を引いて，ゼロからの番号に戻している．
16. L34: xp，yp のループを終える．

演習問題

12.1　新幹線の先頭車両を真横から見た図を描きなさい．

解答例： ソースプログラム Q12_01_Z12_07_bspl_shinkansen.java

付録 A

Fortran ユーザを Java にご招待

Fortran [1]などの手続き型言語に対し，Java などはオブジェクト指向言語[2]といわれる．Fortran などの手続き型言語を熟知していればしているほど，Java などのオブジェクト指向言語は理解しにくいものである．逆に，プログラミング言語を学んだことのない若い人は，抵抗なく理解できるようである．もちろん，Java でも数値計算をする手続きの部分は，手続き型言語のものと同じであるが，両者はまったく別ものであるとして，頭を切り換えて学ぶほうがよいであろう．特に，Fortran の副プログラムに相当する部分は，Java ではクラスとして作られており，それをオブジェクト[3]として実体を取り込んで使うという点が最大な相違であるといえよう．

そもそも，電子計算機は計算を手助けする道具として生まれ，複雑な計算の手続きが指示されて，それを行うのであるから，プログラムはいわゆる手続き型となるのが当然の成り行きである．最初は機械語から始まり，人の労力とミスを少なくするために「せめて数式だけでもそのまま書き，それを計算機に機械語に翻訳させよう」という発想から Fortran などが開発され進歩してきたのである．その進歩の過程でも，常にその時代の計算機の計算速度，メモリ容量などの処理能力の向上に応じて，人の労力を計算機に移していくという方向で，言語仕様が発展してきている．また，計算処理効率を上げる努力は現在でも真剣に続けられている．

一方，オブジェクト指向言語 の仕様は，電子計算機は情報処理の万能な機器としてすでに存在しており，「初めに万能な情報処理機器ありき」という発想から考えられたと極論することも可能であろう．そこで，処理効率より使い勝手に重点が置かれており，重い数値計算にはまったく適していないから，Java と Fortran を使い分ける必要はある．

---

1. FORTRAN は Formula translation（式の翻訳）から作られた名称であり，1957 年に IBM 社によって開発された．
2. Smalltalk という言語がオブジェクト指向言語の最初であるといわれ，計算機の処理能力がかなり向上した 1970 年代に，ゼロックス社によって開発された．
3. Java ではオブジェクトという代わりにインスタンスということもあり，両者は同義語である．本書では，主にインスタンスを使う．

A.1 Fortran と Java の実行

A.1.1 Fortran プログラムの実行

Fortran プログラムの作成から実行までは，標準的には以下のとおりである．

1. ソースプログラム (コード) をテキスト形式で書く．ファイルの拡張子は for, f, f90 などである．プログラムの中で，必要に応じて副プログラムの呼び出しの文を書く．文字コードに依存するが，マシン依存ではない．
2. ソースプログラムをコンパイラにかけ，オブジェクトファイル (モジュール) を作る．コマンドは機種やソフトウェアに依存し，fort, fd, f77, f90 などいろいろある．出力ファイルの拡張子は obj, o などである．マシン依存であり異なった機種では使えない．
3. オブジェクトファイルをリンカにかけ，実行形式ファイル (ロードモジュール) を作る．この際，オブジェクトモジュールに，必要なシステムライブラリやユーザライブラリの副プログラムなどが結合される．ある副プログラムがソースプログラムの中で複数回呼び出されていても，リンクされるものは 1 個である．コンパイルコマンドで，リンクまで行われるものもあるが，link コマンドを使うものもある． 出力ファイルの拡張子は out, exe あるいは拡張子なしである．マシン依存であり異なった機種では使えない．
4. 実行はロードモジュールのファイル名をコマンド名として打ち込むことで行う．

最近は，ソースプログラムの編集画面で，コンパイルから実行までを GUI でできるソフトウェアもあり，それなりに便利である．

A.1.2 Java プログラムの実行

Java プログラムの作成から実行までは，標準的には以下のとおりである．

1. ソースコードをテキスト形式で書く．ファイルの拡張子は java である．プログラムの中で，必要なら他のクラスを継承したり，他のクラスのインスタンス (オブジェクトのこと) を作成する文を書く．ソースコード自体は文字コードに依存するが，マシン依存ではない．
2. ソースコードをコンパイラにかけ，出力クラスファイルを作る．コマンドは javac であり，

   ```
   > javac ソースファイル名.java
   ```

   と打ち込む．ソースコードは，マシンにも文字コードにも依存しないバイトコードといわれる中間言語に翻訳される．出力ファイルの拡張子は class で

ある.

3. 実行のコマンドは java であり,

 > java クラスファイル名 (拡張子 .class は付けない)

 で行う.実行は java コマンドに相当するものが用意されている機械なら,どのような機械でもよく,計算機のウェブブラウザや,携帯電話,家電機器などでも実行できるようにすることが可能であり,実行はバーチャルマシンで行われるという.計算機上で実行される際には,当該クラスに,必要なシステムのライブラリやユーザライブラリのクラスが取り込まれる.同じクラスのインスタンスがソースプログラムの中で異なった名称で複数個作られていたら (呼ばれているのではない),同じ機能をもつインスタンスが複数個存在することになる.もちろん,あるインスタンスの中のメソッド (副プログラムに相当) が複数回呼ばれる場合には,同一のメソッドが使われる.異なる名称 (a, b) で作られた 2 個のインスタンスのメソッド m () やフィールド (変数) x は,a.m () , a.x と b.m () , b.x のようにインスタンス名を使って区別されており,“.” は Fortran の構造体での % の使い方に少し似ている.

最近は,ソースコードの編集画面で,コンパイルと実行までを GUI でできるソフトウェアもあり,それなりに便利である.

まず手始めに,Java の入出力だけのプログラムを書いてみよう.用語の多くは後に説明があるから,とりあえず,わからないことを気にしないで読み進めてほしい.Fortran の write 文で文字列を出力するだけのプログラムは,write 文と end 文の 2 行でよいが,Java では,まずクラスで囲み,その中にメソッドを入れ子にし,その中に print 文を書くから,最低 5 行が必要となる.

A.2 入出力

A.2.1 出力

出力はメソッド System.out.println () で行われる.System クラスはディフォールトでインポートされているから,インポート文なしで使える.引数は文字列であり,文字列,文字,その他の基本データ型を連結演算子 (+) で連結したものである.基本データ型の式は全体を括弧 () で括り,連結する前にそれぞれを 1 個の値としておく必要がある.出力の最後で改行をしない場合は,メソッド System.out.print () を使う.エスケープシークエンスは C 言語と同様である.

出力のプログラム (Print.java) とその説明は以下のとおりである.

Print.java：

```
1 public class Print {
2   public static void main(String[] args) {
3     System.out.print("print は改行なしで，");
4     System.out.println("println は改行あり．");
5   }
6 }
```

1. L1〜6: クラス Print の定義である．
2. L2〜5: メソッド main の定義である．
3. L3: 改行なしのプリントメソッドである．
4. L4: 改行ありのプリントメソッドである．最後に改行を行っている．System クラスはディフォールトでインポートされているから，インポート文はいらない．

A.2.2　入力

入力はあまり簡単ではない．1 バイト入力するだけなら in.read () メソッドを使い，数値のような文字列を入力するには in.readLine () メソッドを使う．入力にエラーがあったときの処理 (例外処理) のために，class の宣言文で，throws IOException を指定しなければならない． そのためには java.io.∗ をインポートしておかなければならない．in.readLine () メソッドでは，文字列が入力されるだけであるが，StringTokenizer クラスを使って，文字列を区切記号ごとに分割して，必要な型変換を行えば数値の入力も可能である．それには java.util.∗ をインポートしておかねばならない．

1 文字を入力するプログラム (Read.java) とその説明は以下のとおりである．

Read.java：

```
1 import java.io.*;
2 public class Read {
3   public static void main(String[] args) throws IOException {
4     System.out.println("1 文字と Enter を入力して下さい．");
5     char c= (char)System.in.read();
6     System.out.println("入力した文字は "+c+" です．");
7   }
8 }
```

1. L1: 入出力の例外処理用の IOException のために java.io.∗ をインポートする．
2. L2〜8: クラス Read の定義である．
3. L3〜7: メソッド main の定義である．例外処理を投げる用意をする．
4. L4: 改行ありのプリントメソッドで文字列の出力をする．
5. L5: 文字型変数 c の宣言を行い，入力メソッドで 1 バイト入力し，文字型に変換するキャスト演算子 (char) を使って文字型に変換した値を c に代入している．
6. L6: 改行ありのプリントメソッドで連結された文字列の出力をする．

数値を入力するプログラム (ReadLine.java) とその説明は以下のとおりである．

ReadLine.java：

```
 1 import java.io.*;
 2 import java.util.*;
 3 public class ReadLine {
 4   public static void main(String[] args) throws IOException {
 5     double a, b, c;
 6     BufferedReader in
 7       = new BufferedReader(new InputStreamReader(System.in));
 8     System.out.println("数値２個と Enter を入力して下さい．");
 9     String s= in.readLine();
10     StringTokenizer t = new StringTokenizer(s);
11     a = Double.parseDouble(t.nextToken());
12     b = Double.parseDouble(t.nextToken());
13     c= a*b;
14     System.out.println("入力した数値の積は "+c+" です．");
15   }
16 }
```

1. L2: L10 にある文字列を区切記号ごとに分割するクラス StringTokenizer のためのインポート文である．
2. L3〜16: クラス ReadLine の定義である．
3. L4〜15: メソッド main の定義である．例外処理を投げる用意をする．
4. L6, 7: BufferReader クラスのインスタンスを作る．
5. L8: 改行ありのプリントメソッドで文字列の出力をする．
6. L9: String s の宣言を行い，入力メソッドで 1 行入力する．s は入力した 1 行を参照している．
7. L10: StringTokenizer クラスのインスタンスを，s を対象として作成する．
8. L11, 12: 最初の区切記号までの文字列を double 型に変換して a に代入し，次の区切記号までの文字列を double 型に変換して b に代入する．
9. L13: a * b を c に代入する．
10. L14: c の値などを出力する．

A.2.3　ファイルからの入力

ファイルからの入出力は文字情報ばかりでなく，画像情報，音声情報などを同一の方法で扱えるようになっているため，もっと複雑である．また，ここで扱うファイル入力は (出力も)，html 文書でウェブに載せることが，セキュリティ上の問題で不可能にされている．

ファイルを入力するプログラム (ReadFile.java) のメソッド (readFile) とその説明は以下のとおりである．

ReadFile.java のメソッド (readFile)：

```
 1   public int readFile(double[] a, String file, String del)
 2       throws IOException {
 3     FileReader fr = new FileReader(file);
 4     BufferedReader br = new BufferedReader(fr);
```

```
 5     StringTokenizer st;
 6     String str;
 7     int n= 0;
 8     while((str=br.readLine())!=null) {
 9       st = new StringTokenizer(str,del);
10       while (st.hasMoreTokens()) {
11         a[n++] = Double.parseDouble(st.nextToken());
12       }
13     }
14     return n;
15   }
```

1. L1〜15: メソッド readFile の定義である．引数の a は入力したデータを代入する配列で，要素数はデータ数より多くしておかねばならない．file は読み込むファイル名，del はデータの区切記号の指定である．データの区切記号のディフォールトは，“ ¥t¥n¥r¥f” 空白，タブ，改行，復帰改行，および用紙送りである．
2. L2: 入出力時のエラー (例外という) 処理のために必要であり，java.io.∗ をインポートしておく必要がある．L3, 4 の FileReader, BufferedReader も同様である．
3. L3: ファイル名を引数にして FileReader クラスのインスタンスを作る．
4. L4: FileReader のインスタンスを引数にして BufferedReader クラスのインスタンスを作る．
5. L5: データ文字列を切り出す StringTokenizer クラスのインスタンス用の名前を定義する．このクラスは java.util.∗ をインポートしておく必要がある．
6. L6: 1 行の文字列を入力する String を定義する．
7. L7: 入力するデータ数を数える int 型変数 n を宣言し，初期値 0 を代入する．
8. L8〜13: ファイルから 1 行読み込み str に代入する．ファイルの最後に来ている場合には，何も読み込めないので，str の値は null となるから，null になるまでループする．
9. L9: 入力した文字列と，区切記号を引数として，StringTokenizer クラスのインスタンスを作る．
10. L10〜12: データがある間はループする．
11. L11: 次のデータ文字列を切り出し，double 型数値に変換し配列 a に代入する．代入が終わったら，n の値をインクリメントする．
12. L14: データ数 n を返す．

A.3　Fortran から見た Java の主な相違点

計算の手続きの書き方は C 言語とほぼ同じであり，Fortran にはない記述方法も多い．以下，羅列的ではあるが，主な相違点をあげてみる．

使用される文字と特殊文字（記号）

使用される文字などは表 A.1 のとおりである．Fortran では英字の大文字と小文字は区別されないが，Java では別物として扱われる．同じスペルでも，クラス名の先頭は大文字にし，メソッド名の先頭は小文字にして区別を明確にしている人もいる．

キーワード（予約語）

Fortran には予約語がないので，例えば真偽値を表すのに .true. .false. のように “.” を使って，名前として定義されるかもしれない true false と区別する必要があるが，Java では true false は真偽値の予約語となっており，名前として使うことはできない．予約語（java ではキーワードといわれる）は全部で約 50 個弱ある（表 A.2 を参照）．

リテラル（定数表現）

Fortran90 の組み込みデータ型に対応し，数字や文字を使ってそのまま表現するデータ（例：1.0, “abc”）である．

表 A.1　Java で使われる文字．

大小英字	A – Z　a – z
数字	0 – 9
特殊文字	_　$　+　-　*　/　%　.　,　:　;
	'　"　=　<　>　(　)　{　}　[　]

表 A.2　Java の予約語．

abstract	boolean	break	byte	case
catch	char	class	const	continue
default	do	double	else	extends
final	finally	float	for	goto
if	implements	import	instanceof	int
interface	long	native	new	package
private	protected	public	return	short
static	strictfp	super	switch	synchronized
this	throw	throws	transient	try
void	volatile	while		

- ❒ 整数の標準は 4 バイトの int 型であり，ほかに 2 バイトの short 型，1 バイトの byte 型と 8 バイトの long 型があり，約 10 桁以下なら int 型となり，10〜19 桁で数字の最後に L または l を付けると long 型となる．そのほか，先頭に 0 を付けると 8 進数，0x を付けると 16 進数となる．
- ❒ 実数型は浮動小数点数といわれ，標準は倍精度の double 型 (例：1.0) であり，単精度は float 型 (例：1.0f) で数字の最後に f を付けて double 型と区別する．
- ❒ 論理型はブール (boolean) 型といわれ，値は true と false の 2 個である．
- ❒ 文字は，文字 1 個を単一引用符 “'” で括った文字 (char) 型 (例：`'a'`) と，0 個以上の文字を二重引用符 “"” で括った文字列 (String) 型 (例：`"abc"`) とが別物として存在する．
- ❒ Java には複素数型はない．

基本データ型

基本データ型は，リテラルにある型から String を除いた種類だけがある．基本データ型の変数の宣言の仕方は，Fortran とほぼ同様である．String は文字列を値としてもつばかりでなく，文字の長さを返すメソッド (length ()) など 47 個のメソッドと 9 個のコンストラクタをもち，基本データ型ではなくクラスである (表 A.3 を参照)．

参照型

Java では，基本データ型以外は参照型といわれ，インスタンス名を宣言し，同時にインスタンスを作って参照[4]を代入するか，他のインスタンスの参照を代入してからでないと使えない．

すでに String は参照型であると述べたが，配列も参照型である．配列の要素は基本データ型ばかりでなくクラスであることもあり，その場合にはそのクラス型の配列となる．2 次元配列は，第 1 添字の配列の各要素として，長さがまちまちな第 2 添字の配列を作ることができる．そこで，記述の仕方もそれなりの特徴があり，Fortran での a (1, 2, 3) は Java では a[0][1][2] となる．添字の値は，Fortran では配列の宣言次

表 A.3　Java の基本データ型．

型	型名
整数	byte，short，int，long
浮動小数点数	float，double
ブール (論理)	boolean
文字	char

4．基本データ型の場合は，変数に値が代入されるが，参照型の場合は，変数に代入されるものは使われるインスタンスの入口 (参照点) である．

第であるが，Java の場合はゼロから始まり C 言語と同じである．

例 1： Class_1 c1; Class_2 c2 = new Class_2 ();

前者は Class_1 という名前のクラスがすでに定義されているときに，そのクラス型のインスタンスの参照を代入することのできる変数名 c1 を宣言し，後者は宣言の後に右辺で Class_2 型のインスタンスを新しく作り，そのインスタンスの参照を c2 に代入する．右辺のメソッド[5] Class_2 () はコンストラクタといわれ，Class_2 型のインスタンスを作るときに使われる．

例 2： String s = “abc”; String s = new String (“abc”);

両者は同じことで，文字列クラスのインスタンスを参照できる変数名 s を宣言し，文字列インスタンス “abc” の参照を代入する．右辺がリテラルの場合は new 演算子は付けない．この場合メソッド s.length () は 3 を返す．

例 3： double[ ] a= {1, 2, 3}; double[ ] a= new double[3];

前者は double 型配列のインスタンスの参照ができる変数名 a を宣言し，double 型配列のインスタンス {1, 2, 3} の参照を代入する．右辺がリテラルの配列定数なので new 演算子は付けない．後者は変数名 a の宣言の後に，要素数が 3 の double 型配列のインスタンスの参照を代入しているが，まだ値は定まっていない．

例 4： double[ ][ ] a= new double[2][3]; double[ ][ ] a={{1}, {2, 3}, {4, 5, 6}};

前者は，第 1 添字の要素数が 2 で，第 2 添字の要素数 3 の長方形の 2 次元配列の宣言と，それが参照するインスタンスの作成をする．後者は，第 1 添字の要素数が 3 で，第 2 添字の要素数が 1，2，3 でまちまちである配列を作成する．後者の第 1 添字の要素数の 3 は a.length で取得することができ，第 1 添字が 1 の要素の第 2 添字の要素数の 2 は a[1].length で取得することができる．

クラス

プログラム単位はクラスである．クラスにはメンバがあり，メンバの種類はフィールド[6]，メソッド[7]と内部クラスである．プログラムの実行が始まる main メソッドを含むクラスのソースファイル名は，クラス名.java でなければならず，外 (この場合は OS) から呼べるように，public 属性[8] を指定しておく．main メソッドには，public と

[5] メソッドは引数がない場合でも () をもつ．

[6] フィールドは変数名，配列名，インスタンス名などであり，値，あるいはインスタンスの参照が代入される．

[7] メソッドは subroutine や function と同じく作業をするプログラム単位である．

[8] Fortran の参照許可指定子は，public と private の 2 個であったが，Java のアクセス修飾子は public，指定なし，protected，private の 4 種類があり，使う対象によって意味合いが異なるので注意が必要である．例： public class test {⋯} をソースファイル test.java に書き，コマンド > javac test.java でコンパイルして，> java test で実行する．test は名称であるから，もちろん他の名称でもよい．

static 属性[9]を指定しておく．

main メソッドを含まないクラスは，main メソッドを含むクラスと同じファイルに書くこともできる．この場合には，コンパイルされると，各クラスがばらばらにされ，各クラスの名称をもつクラスファイルとなり，現ディレクトリに出力される．別のファイルに書いてある場合には，前もってコンパイルを行い，クラスファイルとしておいてもよいし，ファイル名がクラス名と同じであれば，クラスファイルになっていなくても，自動的にファイルが探し出されて，コンパイルされるので，それでもよい．現ディレクトリに置いてあるクラスに対しては，インポート文は必要ない．

一つのクラス内に，型と名称が同じでも，引数が異なるメソッドを，複数定義することができる．Fortran には entry 文があるが，あまり使いやすいものではなかった．

他のクラスの使い方は 2 通りである．

- ❐ クラスの継承——新しく書くプログラムが，すでに存在するプログラムに追加・修正をすればよい場合には，ソースプログラムの先頭で，そのクラスの継承[10]を定義し，それに追加・修正をすることになる．継承される元のクラスは親クラスあるいはスーパークラスといわれ，継承する新しいクラスは子クラスあるいはサブクラスといわれる．子クラスの中で親クラスにあるフィールド名 (例：name) を再定義すると，新しい定義が使われる．親クラスのものを使いたいときには super.name のように予約語の super を前に付けて区別する．子クラスの中で新しいメソッドを作る際に，親クラスにその型と名称が同じメソッドがあり，引数も同じであると，そのメソッドは再定義されることになり，引数が異なれば新たなメソッドが追加されることになる．前者は，メソッドのオーバーライドといわれ，後者はオーバーロードといわれる．引数が異なれば，型と名称が同じメソッドを多数同時に定義しておくことができるが，名称が同じで型が異なるメソッドは両立できない．オーバーライドされた親クラスのメソッド (例：name ()) を使いたいときは，やはり予約語の super をメソッド名の前に付け，super.name () として区別する．
- ❐ クラスのインスタンス化——クラスのもう一つの使い方は，クラスのインスタンス (実体) を作って，その中のフィールド (変数) やメソッドを使う方法である．インスタンスの作成は new 演算子とクラスのコンストラクタ[11]を

---

9. public static void main (String[ ] args) {· · ·} をクラス test の中に書く．static キーワードも属性を指定するものであり，類似なものに final がある．void はメソッドに戻り値がないときに指定し，C 言語と同じである．

10. 継承には extends キーワードを使う．extend は動詞であり，文章であるから 3 人称単数現在のときに付ける s が付いている．implements, throws も同様である．例えば，child という名前のクラスが parent という名前のクラスを継承する場合には，class child extends parent {· · ·} と書く．

11. コンストラクタはクラスの中で定義されており，呼ばれるとそのクラスのインスタンスを作るメソッドである．名前はクラス名と同じで，public 属性のみもっている．引数が空白な場合は，インスタンスの作成時に他のことはしないが，引数に指定があれば，それに応じてインスタンスの内部の初期処理を行う．引数が空白なコンストラクタは，明示的に書かれていなくてもよい．

使って行われる (例：Some_Class s = new Some_Class ()；)．Some_Class という名前のクラスのインスタンスが作られ，それを参照する値が変数名 s に代入される．このインスタンスにあるフィールド名 (例：name) を使うには s.name とし，それがメソッド名ならば s.name () とする．また，new Some_Class () が実引数に使われれば，仮引数側でこのインスタンスを使うことになる．このインスタンスのメンバを使うには，実引数の名称.メンバ名 とする．

メソッドの引数

Fortran とはかなり異なるので注意が必要である．

- 基本データ型——値が渡されるので，メソッド内部での変化は呼び出し元に伝わらない．
- クラス——インスタンスの参照が渡されるので，メソッド内部での変化が呼び出し元に伝わる．
- クラスのメンバを引数に指定することはできない．

演算子など

C 言語とほとんど同じであるが，よく出てくる特徴的なものを拾うことにする．

- べき乗演算子 (a**b) はなく，メソッド Math.pow (a, b) を使う．
- 関係演算子の /= は != である．
- 論理演算子の .not. は !，.and. は &&，.or. は || である．
- 文字列の連結演算子の // は + である[12]．
- Java にはインクリメント ++ とディクリメント −− 演算子がある．a++ は a に 1 を加え，a−− は a から 1 を引く．
- Java の代入演算子は = だけでなく，+=，−=，*=，/=などがあり，x += y は x = x + y，x −= y + z は x = x − (y + z) 等である．
- 三項条件演算子 (〜) ? 〜 : 〜 は，(〜) が真なら ? 以下の値となり，偽なら : 以下の値となる (例：y = (a>b) ? a : b ; では y に大きいほうの値が代入される)．
- キャスト演算子 (型) は変数や値をもつメソッドなどの前に置いて，キャスト演算子に指定した型にその値を変換する (例：double d = 1.0 ; float f = (float) d ;)．

12. 連結される項目は文字と文字列だけではなく，基本データ型の値も文字に変換されて連結される．

制御文

- ❐ do 文の do i=1, n ～ end do は，for 文の for（i=1; i<=n; i++）{～} である．
- ❐ do while 文の do while（～）～ end do は，while 文の while（～）{～} である．
- ❐ Java には dowhile 文 do {～} while（～）がある．
- ❐ select case（～）case（～）～ case（～）～ case default ～ end select は，switch（～）{case ～: ～ case ～: ～ default: ～} である．
- ❐ 制御文の return 文は同じであり，cycle 文は continue 文，exit 文は break 文である．
- ❐ if（～）then ～ else if（～）then ～ else ～ end if は，if（～）{～} else if（～）{～} else {～} である．

コメント

Java は保守がしやすいように設計されているから，その機能が十分生きるように，コメントもきちんと書いておくことが肝要である．

- ❐ // で始まるコメントは，// 以降，行末までがコメントとなる．Fortran の！と同じである．デバッグ用に書かれており，普段は不要な出力文 System.out.println(); などをコメントにする場合にも使われる．
- ❐ /* */ で括られているコメントは，複数行にわたって記述することができる．デバッグ用に使った不要な文をまとめてコメントにする場合にも使われる．
- ❐ /** */ を使って書かれているコメント文は，それをもとに javadoc コマンドを使って仕様書などを html 文書として出力するためにも使われる．文書のレイアウトには htlm 文書のタグを使う (例：
（改行），（上付文字），（下付文字））．

A.4　Math クラスの主なメソッド

数学関係のメソッドをまとめたクラスであり，ディフォールトでインポートされている．また，すべてが static メソッドなので，インスタンスを作らずに直接 Math. メソッド名(); で使うことができる．Fortran の組み込み関数 nint，floor，ceiling の値は整数であるが，Java の rint，floor，ceil の戻り値は double である．Fortran の組み込み手続き call random_number (d) は実数 d か，配列 d に擬似乱数 [0, 1) が戻される．Java の random は乱数が戻り値として返される．Java には円周率 π= Math.PI とオイラー数 e= Math.E がある（表 A.4 を参照）．

表 A.4　Fortran の組み込み関数と Java の Math クラスのメソッドとの対応.

Fortran 関数名	戻り値の型	メソッド名	意　味
abs	double	abs (d)	d の絶対値を返す
abs	float	abs (f)	f の絶対値を返す
abs	int	abs (i)	i の絶対値を返す
abs	long	abs (l)	l の絶対値を返す
sqrt	double	sqrt (d)	d の平方根を返す
sin	double	sin (d)	d のサイン値を返す
cos	double	cos (d)	d のコサイン値を返す
tan	double	tan (d)	d のタンジェント値を返す
exp	double	exp (d)	e の d 乗値を返す
log	double	log (d)	d の自然対数の値を返す
asin	double	asin (d)	d のアークサイン値を返す
acos	double	acos (d)	d のアークコサイン値を返す
atan	double	atan (d)	d のアークタンジェント値を返す
atan2	double	atan2 (d2, d1)	座標 (d1, d2) の偏角 θ を返す
nint	double	rint (d)	d の 4 捨 5 入値を返す
d1**d2	double	pow (d1, d2)	d1 の d2 乗を返す
ceiling	double	ceol (d)	切り上げ数を返す
floor	double	floor (d)	切り下げ数を返す
ナシ	double	toDegrees (d)	ラジアン値 d の度数を返す
ナシ	double	toRadians (d)	度数 d のラジアン値を返す
random_number*	double	random ()	[0, 1) の擬似乱数を返す

* random_number は組み込み手続きである.

A.5　アプリケーションとアプレット

Java で書いたプログラムのコンピュータ上での使い方は 2 通りある．一つ目は，Fortran プログラムのように，コンピュータ上で実行する使い方であり，アプリケーションといわれる．アプリケーションの実行は，main メソッドから始まる．二つ目は，アプレットといわれ，ウェブ上で配信することができる仕掛けが用意されている．そのためには，Applet クラスあるいは swing パッケージを import 文で

```
import java.applet.Applet;
import javax.swing.*;
```

のようにインポートしておき，Applet クラスあるいは JApplet クラスを継承しなければならない．アプレットの実行は，init メソッドから始まる．

本書のために書かれたプログラムは，一つのプログラムでアプリケーション用とアプレット用に使えるような細工がされおり，とても便利である．

次の例は，図 1.1 (p.2) のプログラム Z01_01_cos.java の一部である．このプログラムを変更して新しいプログラムを作る場合には，プログラム名 Z01_01_cos.java のファイル名部分 Z01_01_cos が使われているところをすべて新しいプログラム名，例えば new_program.java の new_program に変更するのがよい．もちろん，メインメソッドなどが含まれているクラスの名称だけを変更すれば実行可能であるけれども，他のプログラムに含まれているさまざまなクラス名と区別しておいたほうがよいからである．

Z01_01_cos.java の一部：

```
 1 /*
 2 <APPLET CODE = "Z01_01_cos.class" WIDTH = 808 HEIGHT = 599></APPLET>
 3 */
 4
 5 import mygks.Gks;
 6 import javax.swing.*;// JApplet, JFrame, JPanel など
 7 import java.awt.*;   // Graphics, Graphicd2D など
 8
 9 public class Z01_01_cos extends JApplet {
10 // アプリケーション用の main を定義し，フレームを作る
11   public static void main(String args[]){
12     Z01_01_cosJFrame fr = new Z01_01_cosJFrame();
13     fr.setTitle("Z01_01_cos");
14 // フレームのサイズは，内部コンテナより，(8,33) 大きくとること
15     fr.setSize(800+8,566+33);
16     fr.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17     fr.setVisible(true);
18   }
19 // アプレット用の init を定義し，パネルを作る
20   public void init() {
21     new Z01_01_cosJPanel(this.getContentPane());
22   }
23 }
24 // アプリケーション用のフレームを定義し，パネルを作る
25 class Z01_01_cosJFrame extends JFrame{
26   public Z01_01_cosJFrame() {
27     new Z01_01_cosJPanel(this.getContentPane());
28   }
29 }
30 // 基本となるパネルを作る．必須
31 class Z01_01_cosJPanel extends JPanel{
32 // 以上は size 以外極めて一般的である．
33 // 以下は例である．
34   public Z01_01_cosJPanel(Container cont){
35     // 図を描くパネルの作成
36     JPanel panel = new JPanel(new BorderLayout()) {
37       public void paint(Graphics g) {
38         super.paint(g);
```

```
39         Z01_01_cosGraph mg= new Z01_01_cosGraph(g);
40         mg.mydraw();
41       }
42     };
43     panel.setSize(800,566);
44 //    panel.setBackground(Color.WHITE);
45     // 図を描くパネルをコンテナに追加する
46     cont.add(panel);
47   }
48 }
49
50 class Z01_01_cosGraph {
51   Gks gk= new Gks();
52   Graphics g;
53
54   public Z01_01_cosGraph(Graphics g) {
55     this.g= g;
56   }
57
58   public void mydraw() {
59     gk.setgks(g);
60 // ここから上をコピーし，<applet> での名前をはじめ，
61 // 17 個すべての名前 GksTemplate をファイルと同じ名前にする．
62
63   中略　図を描く！
64
65   }
66 }
```

1. L1〜3: コメントになっているから，コンパイルの際は無視される．しかし，アプレットを表示するアプレットビューアのコマンド

   > appletviewer ファイル名.java

   で，アプレットビューアに表示する際には解釈される．アプレットビューアを使う前に，このプログラムをコンパイルして，クラスファイルにしておかなくてはならない．
2. L5: 自作のグラフィックスパッケージ mygks のクラス Gks をインポートする．Gks を使わなければ，この行はいらない．
3. L6, 7: グラフィックスとユーザインタフェースのための各種 API が含まれているパッケージの swing と awt をインポートする．
4. L9〜23: swing のクラス JApplet を継承して public class の Z01_01_cos.class を作る．
5. L11〜18: アプリケーションとして使うための main メソッドを作る．
6. L12: 図を表示するためのフレーム (額) を作るインスタンスをクラス Z01_01_cosJFrame（L25〜29) で作る．このクラスは，swing のクラス JFrame を継承している．
7. L13: フレームのタイトルとして Z01_01_cos を指定する．
8. L15: フレームのサイズを 808×599 と指定する．Gks で図を描くパネルの大き

さを L43 で 800×566 と指定するので，額縁の大きさを考慮すると，この数値となる．

9. L16: 額縁の右上にある消去用のボタンをクリックしたら，フレームが消される仕掛けを指定する．
10. L17: フレームに図を表示するように指定する．
11. L20〜22: アプレットとして使うための init メソッドを作る．アプレットの場合には，図を表示するアプレットビューアやウェブのブラウザがフレームを作るから，ここではフレームを作る必要がない．
12. L21: パネルを添付してフレームに乗せるコンテナの取得を引数として，JPanel のインスタンスをクラス Z01_01_cosJPanel 型で作っている．
13. L25〜29: アプリケーション用のフレームを定義するクラスを，JFrame を継承して作る．
14. L26〜28: クラス Z01_01_cosJFrame のコンストラクタであり，ここで，L21 と同じことを行っている．
15. L31〜48: JPanel を継承してパネルを作り，図が描かれたらコンテナに貼り付けると，それで描画が行われる．
16. L36〜42: JPanel 型のインスタンスを作る．パネルにコンポーネントを貼り付けるレイアウトを BorderLayout 形式としている．
17. L37〜41: JPanel のメソッド paint をオーバーライドして，Z01_01_cosGraph クラスの mydraw メソッドでの描画ができるように変更している．
18. L38: まず，親クラスのメソッド paint を実行し，描画の準備をする．
19. L43: 描画するパネルの大きさを 800×566 に指定する．
20. L44: パネルの背景色を白にする．指定しないと灰色である．
21. L46: 図を描くパネルをコンテナに追加する．
22. L50〜66: L39 でインスタンスを作ったクラスの定義である．
23. L51: Gks を使う場合には，そのインスタンスを作る．
24. L52〜56: L37 のメソッド paint の実引数として得られる Graphics 型のインスタンスの参照 g が，L39 でコンストラクタの引数に使われるから，それは L55 の g として L52 の g に与えられ，以後使われる．
25. L58〜65: 図を描くメソッドであり，Gks を使う場合には，setgks メソッドで g の参照値を Gks に渡す．Graphics クラス，Graphics2D クラスあるいは Gks クラスを使って描画をする．
26. L66: Z01_01_cos クラスが終わる．

次は，Z01_01_cos クラスをアプレットとして，ウェブブラウザで見るための HTML 文書である．

```
1 <!-- 図 1.1 cos(x) の級数展開.  Z01_01_cos.java 16/11/04 -->
2 <HTML>
3 <HEAD><TITLE> Z01_01_cos </TITLE></HEAD>
4 <BODY>
5   <APPLET CODE="Z01_01_cos.class" WIDTH=800 HEIGHT=565></APPLET>
6   <P>Copyright(c) 2004;  Wakoh System Laboratory Co., Ltd.</P>
7 </BODY>
8 </HTML>
```

1. L1: タグ <! – – から – – > まではコメントである.
2. L2〜8: タグ <HTML> から </HTML> までが HTML 文である.
3. L3: <HEAD> から </HEAD> まではヘッダ部で，タイトルタグ <TITLE> タイトル </TITLE> でタイトルなどを指定する.
4. L4〜7: <BODY> から </BODY> は HTML 文書の本体である.
5. L5: <APPLET CODE="クラス名"> </APPLET> はアプレットタグであり，CODE 属性にアプレットのクラス名を記し，表示する画面の大きさも指定できる.
6. L6: <P> から </P> はパラグラフタグであり，それまでと異なるパラグラフとして，文章を出力する.

A.6　ライブラリパッケージの作り方と使い方

複数のクラスと 0 個以上のインタフェースをグループ化してパッケージとする.

1. パッケージの作成 —— 下のようにクラスを定義するプログラムの先頭でパッケージ文を指定する.

   package パッケージ名;

   そのプログラムのファイルを

   > javac –d ファイル名

   でコンパイルすると，パッケージ名で指定したディレクトリの下にクラスファイルができる.
2. パッケージのロード —— 下のようにインポート文でインポートしたいクラス名をプログラムの先頭で指定する.

   import パッケージ名. クラス名;
   import パッケージ名.*;

   * を指定すると，パッケージ内のすべてのクラスとインタフェースがインポートされる.
3. import 文で指定されたパッケージは，まず現ディレクトリで探され，なければシステムに指定してあるディレクトリで探される．ほかからもってきたり，自分で作ったりしたパッケージが，現ディレクトリ以外にあれば，環境変数 CLASSPATH にパスを指定しておく必要がある．コンパイルするソースファイルに複数のクラスがある場合には，それらのクラスは，現ディレクトリに作られるから，インポートする必要はない.

java のプログラムで，頻繁に使われる System クラスや Math クラスなどは，インポートしないで使うことができる.

A.7　インタフェースの使い方

Fortran では副プログラムの引数として関数を受け渡しすることができるが，Java ではメソッドを引数として受け渡しすることはできず，そのメソッドが定義されているクラスのインスタンスの参照を受け渡しすることになる．そこで，実引数となるインスタンスのメソッド名と，仮引数のインスタンスのメソッド名を同じにしておかねばならず，少し不便である．しかし，インタフェースとして定義したクラスの抽象メソッドでそのメソッド名を定義しておき，そのインタフェースを実装するクラスで，具体的なメソッドの中身を定義するようにしておくと，そのメソッドを使うクラスに汎用性をもたせることができる．最初に出てきた例は第 4 章のクラス Nibunhou.java のメソッド nibunhou であるから，そのメソッドを使うプログラム NibunhouTest.java に沿って説明する．

プログラム (NibunhouTest.java) とその説明は以下のとおりである．

NibunhouTest.java：

```
 1 public class NibunhouTest {
 2   public static void main(String[] args) {
 3     Function f = new Function(1, 1, -6);  // 2 次関数を x^2+x-6 とする
 4     Nibunhou nb= new Nibunhou();
 5
 6     double x = nb.nibunhou(0.0, 3.0, f);  // [0,3] で解を求める
 7     System.out.println("f(x) = 0 を 2 分法で求める．");
 8     System.out.print("解 x = " + x);
 9     System.out.println("    精度 f(x) = " + f.function(x));
10   }
11 }
```

1. L1〜11: クラス NibunhouTest の定義.
2. L2〜10: メインメソッドの定義.
3. L3: 下で説明する関数のメソッドをもつ Function クラス型のインスタンス f を作成する．クラス Function での関数は 2 次式であり，2 次式の係数はコンストラクタの引数で与えられる．引数は (1, 1, −6) であるから，関数は $x^2 + x - 6$ である.
4. L4: クラス Nibunhou 型のインスタンス nb を作成する.
5. L6: クラス Nibunhou のメソッド nibunhou を使って，範囲 (0.0, 3.0) で関数 f の根を探し，x に代入する.
6. L7〜9: 根の出力をし，求められた根の値を使って関数の計算をして，精度の目安として出力する.

プログラム (Function.java) とその説明は以下のとおりである．

Function.java：

```
 1 class Function implements MyFunction {
 2   private double a, b, c;
 3   public Function(double a, double b, double c) {  // コンストラクタ
 4     this.a = a;
 5     this.b = b;
 6     this.c = c;
 7   }
 8   public double function(double x) {  // 関数
 9     return (a*x + b)*x + c;
10   }
11 }
```

1. L1〜11: 下で説明するインタフェース MyFunction を実装するクラス Function の定義である．
2. L2: コンストラクタ Function（L3〜7）とメソッド function (L8〜10）で共通に使うフィールド (変数）a, b, c の宣言．
3. L3〜7: オーバーロードするコンストラクタ Function の定義．
4. L4〜6: コンストラクタ Function の引数 a, b, c の値を，L2 で宣言した a, b, c に代入する．キーワード this はこのクラスを指す．
5. L8〜10: L2 で宣言した a, b, c を係数とする，2 次式の定義で，メソッド function をオーバーライドする．

インタフェース (MyFunction.java) とその説明は以下のとおりである．

MyFunction.java：

```
1 public interface MyFunction {
2   public double function(double x);
3 }
```

1. L1〜3: インタフェース MyFunction の定義である．
2. L2: 抽象メソッド function の定義．メソッドの型と引数 x のみが定義されており，MyFunction を実装するクラスの中でオーバーライドされ，実体はそのクラスの中で定義される．

付録 B

mygks メモ

本書のプログラムでは Java のグラフィックス機能を使って作ったパッケージ mygks を使っている．まだ完熟したものではないので，β 版として一部を提供する．表 B.1 に基本的な図形出力とその属性指定などのメソッドを，表 B.2 にその属性をまとめた．

プログラム (Z01_01_cos.java) のクラス (Z01_01_cosGraph) に沿って，簡単な解説を行う．付録 A のプログラム Z01_01_cos.java の解説と併せて理解してほしい．

Z01_01_cos.java のクラス (Z01_01_cosGraph)：

```
1 class Z01_01_cosGraph {
2   Gks gk= new Gks();
3   Graphics g;
4
5   public Z01_01_cosGraph(Graphics g) {
6     this.g= g;
```

表 B.1　mygks の基本的な図形出力とその属性を指定するメソッド．i は表 B.2 の属性の番号，d は線幅とマーカの倍率あるいは文字の大きさ，x，y は座標，xa，ya は座標の配列，n は描画に使う点の数である．また s は文字列である．

	描画	種類	大きさ	色
線	gpl (n，xa，ya)	gsln (i)	gslwsc (d)	gsplci (i)
マーカ	gpm (n，xa，ya)	gsmk (i)	gsmksc (d)	gspmci (i)
文字	gtx (x，y，s)		gschh (d)	gstxci (i)

表 B.2　mygks の図形出力の色，線種，マーカ属性．

i	0	1	2	3	4	5	6	7
色	黒	赤	緑	黄	青	紫	空	白
線種		実線	点線	破線	鎖線	一点鎖線	二点鎖線	三点鎖線
マーカ		·	+	*	○	×		

```
 7   }
 8
 9   public void mydraw() {
10     gk.setgks(g);
11
12     double xscale=  2*Math.PI;;
13     double wxmin= -2.0*xscale, wxmax= 2.0*xscale;  //  x 世界座標範囲
14     double xmin= -1.0*xscale, xmax= 1.0*xscale;    //  x グラフの範囲
15
16     double yscale= 1.0;
17     double wymin= -4.0*yscale, wymax= 4.0*yscale;  //  y 世界座標範囲
18
19     String free= "fr";
20     gk.gopen(wxmin,wxmax, wymin,wymax, free);
21
22     double[][] win= gk.gqwindow();  //  ウィンドウの定数の取得
23
24     double dsx= win[2][0];  //  x 量/cm
25     double dsy= win[2][1];  //  y 量/cm
26
27     gk.gsln(1);          //  線種を実線に指定
28     gk.gslwsc(3.0);      //  線の太さを 3 倍に指定
29     gk.gsplci(0);        //  線の色を黒に指定
30     gk.gsmk(1);          //  点印を指定
31     gk.gsmksc(2.0);      //  マーカサイズを 2 倍指定
32     gk.gspmci(0);        //  マーカの色を黒に指定
33     double h=gk.gqchh();//  文字の高さを取得
34     gk.gschh(1.5*h);     //  文字の高さを 1.5 倍に指定
35     gk.gstxci(0);        //  文字の色を黒に指定
36
37     int nmaxx= 101;      //  配列要素数
38     double[] x= new double[nmaxx];  //  x 座標用配列
39     double[] y= new double[nmaxx];  //  cos 関数用配列
40
41     int nkind= 5;
42     double[][] y1= new double[nkind][nmaxx];//  近似関数用配列
43
44 //  x 座標,  cos 関数, 近似関数の配列データの作成
45     for (int i=0; i<nmaxx; i++) {
46       double w= x[i]= (-1.0+i*2.0/(nmaxx-1))*xmax;
47       y[i]= Math.cos(w);
48       y1[0][i]= 1.0;
49       y1[1][i]= y1[0][i] - w*w/2;
50       y1[2][i]= y1[1][i] + w*w*w*w/24;
51       y1[3][i]= y1[2][i] - w*w*w*w*w*w/720;
52       y1[4][i]= y1[3][i] + w*w*w*w*w*w*w*w/40320;
53     }
54
55 //  座標系の表示
56   //  座標軸の表示
57     gk.gVector(xmin*1.3,0.0, xmax*1.3,0.0, 0.3);//  x 座標
```

```
58      gk.gVector(0.0,wymin, 0.0,wymax, 0.3);      //  y 座標
59    //  x 軸上目盛り
60      gk.gslwsc(2.0);  //  線の太さを 2 倍に指定
61      gk.gjoin(xmin,0.0, xmin,dsy*0.3);      //  目盛り線
62      gk.gjoin(xmin/2,0.0, xmin/2,dsy*0.3);  //  目盛り線
63      gk.gjoin(xmax/2,0.0, xmax/2,dsy*0.3);  //  目盛り線
64      gk.gjoin(xmax,0.0, xmax,dsy*0.3);      //  目盛り線
65    //  x 軸上目盛り文字
66      gk.gtx(xmin-dsx*0.8,-dsy*0.8,"-2π");
67      gk.gtx(xmin/2-dsx*0.5,-dsy*0.8,"-π");
68      gk.gtx(dsx*0.2,-dsy*0.8,"0");
69      gk.gtx(xmax/2-dsx*0.4,-dsy*0.8,"π");
70      gk.gtx(xmax-dsx*0.5,-dsy*0.8,"2π");
71    //  y 軸上目盛り
72      gk.gjoin(0.0,2.0, 0.25,2.0);    //  目盛り線
73      gk.gjoin(0.0,-1.0, 0.25,-1.0);  //  目盛り線
74      gk.gjoin(0.0,-2.0, 0.25,-2.0);  //  目盛り線
75    //  y 軸上目盛り文字
76      gk.gtx(-dsx*0.5,2.0-dsy*0.25,"2");
77      gk.gtx(-dsx*1.0,-1.0-dsy*0.25,"-1");
78      gk.gtx(-dsx*1.0,-2.0-dsy*0.25,"-2");
79  //  関数の表示
80      gk.gpl(nmaxx,x,y);              //  cos 関数グラフ
81  //  近似関数を nkind=5 種類表示
82      for (int i=0; i<nkind; i++) {
83        gk.gsln(i+1);           //  線種を指定
84        gk.gsplci(i+1);         //  線の色を指定
85        gk.gpl(nmaxx,x,y1[i]);  //  近似関数グラフ
86      }
87    }
88  }
```

1. L1〜88: Z01_01_cos.java のクラス Z01_01_cosGraph の定義である．
2. L2: クラス Gks のインスタンスを作り，その参照を gk に代入する．
3. L3: Graphics 型のインスタンスを参照する変数 g を宣言する．
4. L5〜7: コンストラクタであり，外部でインスタンスが作られるとき，引数で Graphics 型のインスタンスを取得し，その参照を L3 の g に代入する．
5. L9〜87: 図を描くメソッド mydraw の定義である．
6. L10: Gks を初期化する．setgks は規格外の追加機能である．
7. L12〜20: 画面内に描画する世界座標の範囲を指定する．
8. L20: gopen で世界座標の x 軸の範囲を wxmin から wxmax とし，y 軸の範囲を wymin から wymax とする．第 5 引数を “fr” とすると，縦横の単位は独立であるが，“fx” とすると同じ単位となる．gopen は規格外の追加機能である．
9. L22〜25: ウィンドウの定数を取得する．画面上約 1 cm 当たりのピクセル数を dsx と dsy に代入する．文字列や目盛りの位置の指定などに使う．
10. L27〜35: 国際規格の GKS の関数と同じ名称と機能のメソッドである．線，マーカ，文字の属性を設定するメソッドは表 B.1 のとおりで，属性は表 B.2 のとおりである．

11. L57, 58: 以後 #n で引数の番号とする．点 (#1, #2) から点 (#3, #4) までのベクトルを使って座標軸を描く．#5 はベクトルの鏃の大きさを cm で指定する．線の属性を使う．規格外の追加機能である．
12. L61, 64: 点 (#1, #2) から点 (#3, #4) までを線で結び，x 軸上の目盛りを描く．線の属性を使う．規格外の追加機能である．
13. L66〜70: 点 (#1, #2) を文字列の左下の座標として，#3 の文字列 (x 軸上目盛り文字) を出力する．
14. L72, 74: y 軸上の目盛りを描く．
15. L76〜78: y 軸上目盛り文字を出力する．
16. L80: 配列 x, y の各要素の組み合わせを座標点として，nmaxx 点を折れ線で結ぶ．
17. L82〜86: それぞれ線種，線の色を変えて近似関数のグラフを描く．

付録 C

ソース&クラスリスト

本書で扱っているプログラムのソースコードのリストである．備考欄に class と書かれているプログラムには，同名のクラスファイルがパッケージ mypackage にある．html と書かれているファイルは html 文書である．

これらの Java プログラムは

http://java.sun.com/j2se/1.4.2/ja/download.html

の開発環境で作られたが，2004 年 9 月 30 日には，新しいバージョン

http://java.sun.com/j2se/1.5.0/ja/download.html

が提供されいる．両開発環境は，それぞれのウェブページから無償でダウンロードすることができる．

このリストにあるファイルと，パッケージ mygks はウェブページ

http://www.tdupress.jp/

（［メインメニュー］→［ダウンロード］→［Java で学ぶ数値解析］）

にあり，本書購読者は無償でダウンロードすることができる．

ファイル名	節／項	目　的	備考
Bspl.java	3.5.1 項	B スプライン	class
BsplTest.java	3.5.1 項	B スプラインのテスト	
Bspl_Closed.java	12.3.3 項	スプラインによる 2 次元閉曲線	class
Bspl_ClosedTest.java	12.3.3 項	スプラインによる 2 次元閉曲線テスト	
Bspl_Open.java	12.3.1 項	スプラインによる 2 次元閉曲線	class
Bspl_OpenTest.java	12.3.1 項	スプラインによる 2 次元閉曲線テスト	
Bspl_Smooth.java	8.3.2 項	B スプラインによる平滑化	class
Bspl_SmoothTest.java	8.3.2 項	B スプラインによる平滑化テスト	
Bspline_Interpolation.java	3.5.1 項	B スプラインを使った内挿	class
Bspline_InterpolationTest.java	3.5.1 項	B スプラインを使った内挿テスト	
Eigen_Power.java	9.2.4 項	累乗法による固有値問題解法	class
Eigen_PowerTest.java	9.2.4 項	累乗法による固有値問題解法テスト	
Eigen_QR.java	9.2.5 項	QR 法による固有値問題解法	class
Eigen_QRTest.java	9.2.5 項	QR 法による固有値問題解法テスト	

ファイル名	節／項	目　的	備考
Fast_Fourier_Transform.java	10.3 節	高速フーリエ変換	class
Fast_Fourier_TransformTest.java	10.3 節	高速フーリエ変換テスト	
FindZero.java	4.6 節	複数解の解法汎用プログラム	class
FindZeroTest.java	4.6 節	複数解の解法汎用プログラムテスト	
Fourier_Transform.java	10.2 節	フーリエ変換	class
Fourier_TransformTest.java	10.2 節	フーリエ変換テスト	
Gauss_Hakidashihou.java	7.1.2 項	掃き出し法による連立方程式の解法	class
Gauss_HakidashihouTest.java	7.1.2 項	掃き出し法による連立方程式の解法テスト	
Gauss_Hermit.java	5.3.4 項	ガウス・エルミートの積分公式	class
Gauss_HermitTest.java	5.3.4 項	ガウス・エルミートの積分公式テスト	
Gauss_Laguerre.java	5.3.3 項	ガウス・ラゲールの積分公式	class
Gauss_LaguerreTest.java	5.3.3 項	ガウス・ラゲールの積分公式テスト	
Gauss_Legendre.java	5.3.1 項	ガウス・ルジャンドルの積分公式	class
Gauss_LegendreTest.java	5.3.1 項	ガウス・ルジャンドルの積分公式テスト	
Gauss_Shoukyohou.java	7.1.1 項	消去法による連立方程式の解法	class
Gauss_ShoukyohouTest.java	7.1.1 項	消去法による連立方程式の解法テスト	
Gauss_Tchebycheff.java	5.3.2 項	ガウス・チェビシェフの積分公式	class
Gauss_TchebycheffTest.java	5.3.2 項	ガウス・チェビシェフの積分公式テスト	
Gyaku_2jikansuu.java	4.5 節	逆 2 次関数法による解法	class
Gyaku_2jikansuuTest.java	4.5 節	逆 2 次関数法による解法テスト	
H03_01_lagrange.java	3.1 節	表 3.1	
H03_02_NewtonInt.java	3.1 節	表 3.2	
H08_01_tchebycheff.java	8.2 節	表 8.1	
Hasamiuchi.java	4.3 節	挟み撃ち法による解法	class
HasamiuchiTest.java	4.3 節	挟み撃ち法による解法テスト	
Hermit.java	3.4 節	エルミートの補間法	class
HermitTest.java	3.4 節	エルミートの補間法テスト	
Integral_Simpson.java	5.2.2 項	シンプソンの積分公式	class
Integral_SimpsonTest.java	5.2.2 項	シンプソンの積分公式テスト	
Jacobi.java	9.2.3 項	ヤコビ法	class
JacobiTest.java	9.2.3 項	ヤコビ法テスト	
Lag.java	3.1 節	ラグランジュの補間法	class
LagTest.java	3.1 節	ラグランジュの補間法テスト	
Lagrange.java	3.2 節	ラグランジュ補間の汎用プログラム	class
LagrangeTest.java	3.2 節	ラグランジュ補間の汎用プログラムテスト	
LU_Bunkai.java	7.1.3 項	LU 分解による連立方程式の解法	class
LU_BunkaiTest.java	7.1.3 項	LU 分解による連立方程式の解法テスト	
LU_Gyouretsushiki.java	7.2.2 項	LU 分解による行列式の計算	class
LU_GyouretsushikiTest.java	7.2.2 項	LU 分解による行列式の計算テスト	
Maximum.java	2.2 節	最大値	class
MaximumTest.java	2.2 節	最大値テスト	
Merge.java	2.10 節	マージ	class
MergeTest.java	2.10 節	マージテスト	
MyFunction.java	A.7 節	戻り値 double のインタフェース	class
MyFunction1.java	6.4.1 項	戻り値 double[ ] のインタフェース	class
MyFunction2.java	8.2 節	戻り値 double[ ][ ] のインタフェース	class

ファイル名	節／項	目　的	備考
NewtonInt.java	3.3 節	ニュートンの補間法	class
NewtonIntTest.java	3.3 節	ニュートンの補間法テスト	
NewtonRaphson.java	4.4.1 項	ニュートン・ラフソン法による解法	class
NewtonRaphsonTest.java	4.4.1 項	ニュートン・ラフソン法による解法テスト	
Nibunhou.java	4.2 節	2 分法による解法	class
NibunhouTest.java	4.2 節	2 分法による解法テスト	
NijiHouteishiki.java	4.1.1 項	2 次方程式の解法	class
NijiHouteishiki_Answer.java	4.1.1 項	2 次方程式の解のクラス	class
NijiHouteishikiTest.java	4.1.1 項	2 次方程式の解法テスト	
Print.java	A.2.1 項	出力サンプルプログラム	
Random_Exp.java	11.2.1 項	標準指数分布乱数	class
Random_ExpTest.java	11.2.1 項	標準指数分布乱数テスト	
Random_Function.java	11.2.5 項	表読取り法による関数分布乱数	class
Random_FunctionTest.java	11.2.5 項	表読取り法による関数分布乱数テスト	
Random_Lorentzian.java	11.2.2 項	ローレンツ型分布乱数	class
Random_LorentzianTest.java	11.2.2 項	ローレンツ型分布乱数テスト	
Random_Normal.java	11.2.3 項	正規分布乱数	class
Random_NormalTest.java	11.2.3 項	正規分布乱数テスト	
Random_Sphere.java	11.2.4 項	球面一様分布乱数	class
Random_SphereTest.java	11.2.4 項	球面一様分布乱数テスト	
Read.java	A.2.2 項	1 文字入力サンプルプログラム	
ReadFile.java	A.2.2 項	ファイルからの数値入力	class
ReadFileTest.java	A.2.2 項	ファイルからの数値入力テスト	
ReadLine.java	A.2.2 項	数値入力サンプルプログラム	
RungeKutta4.java	6.4.1 項	ルンゲ・クッタ法 4 次公式	class
RungeKutta4Test.java	6.4.1 項	ルンゲ・クッタ法 4 次公式テスト	
Saishou_Jijou_Hou.java	8.1.1 項	最小二乗法	class
Saishou_Jijou_HouTest.java	8.1.1 項	最小二乗法テスト	
SanjiHouteishiki.java	4.1.2 項	3 次方程式	class
SanjiHouteishikiTest.java	4.1.2 項	3 次方程式テスト	
Savitzky_Golay.java	8.3.1 項	サビツキー・ゴーレイ法による平滑化	class
Savitzky_GolayTest.java	8.3.1 項	サビツキー・ゴーレイ法による平滑化テスト	
Shuffle.java	2.1 節	シャッフル	class
ShuffleTest.java	2.1 節	シャッフルテスト	
Sort_Bubble.java	2.3 節	バブルソート	class
Sort_BubbleTest.java	2.3 節	バブルソートテスト	
Sort_Comb.java	2.4 節	櫛ソート	class
Sort_CombTest.java	2.4 節	櫛ソートテスト	
Sort_Heap.java	2.9 節	ヒープソート	class
Sort_HeapTest.java	2.9 節	ヒープソートテスト	
Sort_Insert.java	2.6 節	挿入ソート	class
Sort_InsertTest.java	2.6 節	挿入ソートテスト	
Sort_MergeSort.java	2.11 節	マージソート	class
Sort_MergeSortTest.java	2.11 節	マージソートテスト	
Sort_Quick.java	2.8 節	クイックソート	class
Sort_QuickTest.java	2.8 節	クイックソートテスト	

ファイル名	節／項	目　的	備考
Sort_Select.java	2.5 節	選択ソート	class
Sort_SelectTest.java	2.5 節	選択ソートテスト	
Sort_Shell.java	2.7 節	シェルソート	class
Sort_ShellTest.java	2.7 節	シェルソートテスト	
SphericalBessel.java	1.4 節	球ベッセル関数 ($l \leq 20$)	class
SphericalBesselTest.java	1.4 節	球ベッセル関数 ($l \leq 20$) テスト	
SphericalBessel_4.java	1.4 節	球ベッセル関数 ($l \leq 4$)	class
SphericalBessel_4Test.java	1.4 節	球ベッセル関数 ($l \leq 4$) テスト	
Spline3_IntNP.java	3.5.2 項	3 次の非周期スプライン補間	class
Spline3_IntNPTest.java	3.5.2 項	3 次の非周期スプライン補間テスト	
Spline3_IntP.java	3.5.2 項	3 次の周期スプライン補間	class
Spline3_IntPTest.java	3.5.2 項	3 次の周期スプライン補間テスト	
Tchebycheff.java	8.2 節	チェビシェフ多項式近似	class
TchebycheffTest.java	8.2 節	チェビシェフ多項式近似テスト	
Z01_01_cos.java	1.1 節	図 1.1	
Z01_02_kizami.java	1.2 節	図 1.2	
Z01_03_ketaochi.java	1.3 節	図 1.3	
Z01_04_bessel.java	1.4 節	図 1.4	
Z01_05_j_rec.java	1.4 節	図 1.5	
Z01_06_marume.java	1.5 節	図 1.6	
Z02_01_shuffle.java	2.1 節	図 2.1	
Z02_02_maximum.java	2.2 節	図 2.2	
Z02_03_bubble_exp.java	2.3 節	図 2.3	
Z02_04_bubble.java	2.3 節	図 2.4	
Z02_05_insert_exp.java	2.6 節	図 2.5	
Z02_06_sort_quick_exp.java	2.8 節	図 2.6	
Z02_07_sort_heap_exp.java	2.9 節	図 2.7	
Z02_08_merge.java	2.10 節	図 2.8	
Z02_09_merge_exp.java	2.10 節	図 2.9	
Z02_10_sort_merge_exp.java	2.11 節	図 2.10	
Z03_01_lagrange.java	3.1 節	図 3.1	
Z03_02_lagrange.java	3.1 節	図 3.2	
Z03_03_lagrange.java	3.2 節	図 3.3	
Z03_04_NewtonInt.java	3.3 節	図 3.4	
Z03_05_hermit.java	3.4 節	図 3.5	
Z03_06_deBoor_Cox_exp.java	3.5.1 項	図 3.6	
Z03_07_Bspline_Interpolation.java	3.5.1 項	図 3.7	
Z03_08_Bspline_Extended.java	3.5.1 項	図 3.8	
Z03_09_Bspline_Heart.java	3.5.1 項	図 3.9	
Z03_10_Spline3_IntNP.java	3.5.2 項	図 3.10	
Z03_11_Spline3_IntP.java	3.5.2 項	図 3.11	
Z04_01_nibunhou_exp.java	4.2 節	図 4.1	
Z04_02_hasamiuchihou.java	4.3 節	図 4.2	
Z04_03_newton_raphson_exp.java	4.4.1 項	図 4.3	
Z04_04_hanpuku_exp.java	4.4.2 項	図 4.4	
Z04_05_gyaku_2jikansuu_exp.java	4.5 節	図 4.5	

ファイル名	節／項	目　的	備考
Z04_06_FindZero.java	4.6 節	図 4.6	
Z05_01_kukei.java	5.1 節	図 5.1	
Z05_02_daikei_simp.java	5.2.2 項	図 5.2	
Z05_03_daikei_simp.java	5.2.2 項	図 5.3	
Z05_04_gauss_exp.java	5.3.1 項	図 5.4	
Z05_05_gauss10_exp.java	5.3.1 項	図 5.5	
Z05_06_gauss_tchebycheff_exp.java	5.3.2 項	図 5.6	
Z05_07_gauss_laguerre_exp.java	5.3.3 項	図 5.7	
Z05_08_gauss_hermit_exp.java	5.3.4 項	図 5.8	
Z06_01_bibun_shiki_exp.java	6.1 節	図 6.1	
Z06_02_runge_kutta_exp.java	6.2.2 項	図 6.2	
Z06_03_orbit_pro.java	6.4.2 項	図 6.3	
Z06_04_Rkg_H.java	6.5 節	図 6.4	
Z06_05_Rkg_H_1s2s.java	6.5 節	図 6.5	
Z07_01_pyramid.java	7.1.2 項	図 7.1	
Z07_02_gyouretsushiki_exp.java	7.2.2 項	図 7.2	
Z08_01_Saishou_Jijou_Hou.java	8.1.1 項	図 8.1	
Z08_02_SJH_2jishiki.java	8.1.3 項	図 8.2	
Z08_03_tchebycheff.java	8.2 節	図 8.3	
Z08_04_savitzky_golay_2ji.java	8.3.1 項	図 8.4	
Z08_05_savitzky_golay.java	8.3.1 項	図 8.5	
Z08_06_Bspl_Smooth_2ji.java	8.3.2 項	図 8.6	
Z08_07_Bspl_Smooth.java	8.3.2 項	図 8.7	
Z08_08_Bspl_Smooth_peak.java	8.3.2 項	図 8.8	
Z08_09_FFT_Smooth.java	8.3.3 項	図 8.9	
Z09_01_ZahyouHenkan.java	9.1 節	図 9.1	
Z09_02_JacobiDaen.java	9.1 節	図 9.2	
Z09_03_JacobiDaenMen.java	9.2.3 項	図 9.3	
Z10_01_fourier_sin.java	10.1 節	図 10.1	
Z10_02_fourier_cos.java	10.1 節	図 10.2	
Z10_03_gauss.java	10.3 節	図 10.3	
Z10_04_FFTC_gauss.java	10.3 節	図 10.4	
Z10_05_polar.java	10.4 節	図 10.5	
Z10_06_polar_dr.java	10.4 節	図 10.6	
Z10_07_fourier_H1s.java	10.4 節	図 10.7	
Z10_08_conv_Cu.java	10.5.1 項	図 10.8	
Z10_09_fourier_Cu.java	10.5.1 項	図 10.9	
Z10_10_deconv_Cu.java	10.5.2 項	図 10.10	
Z11_01_monte.java	11.1 節	図 11.1	
Z11_02_exp.java	11.2.1 項	図 11.2	
Z11_03_exp_exp.java	11.2.1 項	図 11.3	
Z11_04_lorentzian.java	11.2.2 項	図 11.4	
Z11_05_seikibunpu.java	11.2.3 項	図 11.5	
Z11_06_sphere.java	11.2.4 項	図 11.6	
Z11_07_rand_table.java	11.2.4 項	図 11.7	
Z11_08_buffon.java	11.3.1 項	図 11.8	

ファイル名	節／項	目　的	備考
Z11_09_MtAsama.java	11.3.2 項	図 11.9	
Z11_10_parabola.java	11.3.2 項	図 11.10	
Z12_01_truncated.java	12.1 節	図 12.1	
Z12_02_Bspline_3.java	12.2 節	図 12.2	
Z12_03_Bspline_Mult.java	12.2 節	図 12.3	
Z12_04_bspl_openline.java	12.3.1 項	図 12.4	
Z12_05_Bspl_Riesenfeld.java	12.3.1 項	図 12.5	
Z12_06_Bspl_Riesenfeld_Mult.java	12.3.2 項	図 12.6	
Z12_07_bspl_closed_heart.java	12.3.2 項	図 12.7	
Z01_01_cos.html	1.1 節	図 1.1	html

演習問題解答（プログラム例）

ファイル名	演習問題
Q01_01_Z01_01_cos.java	1.1
Minimum.java，MinimumTest.java	2.1
Sort_Bubble_d.java，Sort_Bubble_dTest.java	2.2
Sort_Quick_d.java，Sort_Quick_dTest.java	2.3
Q03_01_Z03_03_lagrange.java	3.1
Q06_01_Z06_03_shindou.java	6.1
Q08_01_Z08_02_SJH_2jishiki.java	8.1
Q10_01_Z10_03_delta.java	10.1
Q11_01_Z11_01_janken.java	11.1
Q11_02_Z11_01_janken.java	11.2

あとがき

著者が最初に電子計算機に触れたのは，東大の理学部物理学科に進学した 1959 年のことである．当時，高橋・後藤研で発明されたパラメトロン素子を使った試作機 PC-1 (Parametron Computer 1) での実習が，学生実験の科目に組み込まれていたからである．メモリは 512 語 (1 語＝ 3 バイトに相当) しかなく，テレックス通信用に使われていた，タイプライタと紙テープの読み取り・打ち出し装置を合わせた機器だけが付いていた．プログラムは機械向き言語で書かれ，紙テープに鑽孔 (穴あけ) して読み取り装置にかけると，前もってメモリに載せてあるイニシャルローダーがプログラムを 2 進数に変換してメモリに載せ，それが実行されるものであった．イニシャルローダーは，現在の OS の先祖であり，入力と出力を処理する機能をもっており，512 語のメモリのうち約 450 語ほどを使っていたから，利用者が使えるメモリは 60 語程度しかなかった．それでも，平方根や三角関数の計算，あるいは簡単な式のシンプソン積分などは工夫次第で収めることができたので，学生実験レポートのテーマとなっていた．

1960 年代になると，国産の電子計算機も実用の段階に入り，前半は機械向き言語であったが，後半になるとコンパイラ言語の ALGOL や FORTRAN が使えるようになってきた．この頃になると，平方根はハード素子で，また三角関数などはソフトウェアで提供されるようになったが，少し複雑な関数や手続きなどの副プログラムは，すべて自分で書かなければならなかった．

このような経緯から，この本で取り扱うプログラムの多くは，当初は機械向き言語，ALGOL，FORTRAN 等いろいろなプログラミング言語で書かれ，また，書き直されたものがほとんどである．もちろん，Java 言語のものはつい最近書き直したものである．

周知のごとく，IT 技術の発展は計算機のハードウェアとソフトウェアにとどまらず，マルチメディアにまで及び，多くの言語でグラフィックスを扱えるようになった．数値解析では，一見，グラフィックスを使わなくても良さそうであるが，デバッグの段階で使用すると，その効果は絶大である．数百から数千ステップにも及ぶ数値計算のプログラムを作るときのデバッグは，まさに砂を噛む思いである．特に，あるテーマに関して，世の中で最初のプログラムを書くときは，結果を比較するデータがないわけであるから，最終結果が出るまでは手探りの状態である．例えば，物理の理論計算などでは，極端な場合，最後の一つの数値を，対応する実験値と照らし合わせることになる．実験誤差の範囲や理論計算の精度がはっきりしていたり，いろいろな条件

下での実験値と比較できたりすれば，プログラムの正誤の判断も可能であるが，そうでなければ他の計算手段(60 年代では手回し計算機)で一通りの計算をして，デバッグする必要があった．その過程で，計算機の出力を方眼紙の上に手でプロットして考察することがよくあった．1970 年代に入り XY プロッタが使えるようになると，早速プロッタを利用することになった．当初使えた命令は，ペンの現在位置を原点とする命令と，指定した位置までペンを上げたまま，あるいは下げたまま移動しなさいという命令だけであった．そこで，初期化は，ペンを x あるいは y の負の方向に大きく移動させ，装置の端に押し付けて，そこを原点とすることであった．それでも，便利な副プログラムを作れば，楽に使いこなすことはできたが，紙の上に描くのであるから時間がかかり，あまり便利なことではなかった．

1980 年代に入り，CRT (テレビ画面) が使えるようになると，グラフィックスの国際規格の GKS 等が制定されたりして，状況は一転し，現在のアニメーションへと発展することになった．当時，GKS 等のグラフィックスパッケージは非常に高価なものであったが，計算センターなどの汎用大型計算機では FORTRAN コンパイラなどとともに導入され，身近で使いやすいものであった．しかし，1990 年代になり，ワークステーションやパソコン等へのダウンサイジングが進むと，個々のプラットホームごとでのパッケージの導入が難しくなり，GKS を使った過去の財産が使えなくなるという危機に直面することとなった．そこで，著者は PC の Windows のグラフィックス機能を使って，Fortran で呼べる GKS の小さなサブセットとなるサブルーチンパッケージを作った [1]．また，Java が一般的に使えるようになってきたので，すでに述べたように Java のグラフィックス機能を使って，GKS タイプの図形出力などのクラスを作成し，パッケージ mygks として提供することになった．

最後に，本書が，理工系の若い学生はもとより，IT 産業に従事している科学技術研究者が，ソフトウェアの根幹の一つをなしている数値解析の原理を学ぶことに役立ち，また，昭和の手続型言語の時代に日本の高度成長を支えてきた中高年のエキスパートにとって，平成のオブジェクト指向言語の一つである Java に親しむきっかけとなり，元気を取り戻してもらうことに少しでも役立つならば，それは本当に望外の喜びであります．

2005 年 3 月

著者しるす

参考文献

[1] 和光信也,「コンピュータで見る固体の中の電子」, 講談社, 1993.
[2] 宇野利雄,「計算機のための数値計算」, 朝倉書店, 1963.
[3] 一松信, 森口繁一, 山内二郎 共編,「電子計算機のための数値計算法 I」, 培風館, 1981.
[4] 一松信, 森口繁一, 山内二郎 共編,「電子計算機のための数値計算法 II」, 培風館, 1981.
[5] 一松信, 森口繁一, 山内二郎 共編,「電子計算機のための数値計算法 III」, 培風館, 1981.
[6] 林英輔, 安井勝, 高橋健,「数値計算」, 森北出版, 1984.
[7] 三井田惇郎, 須田宇宙,「数値計算法」, 森北出版, 2000.
[8] 小国力,「Fortran95, C & Java による新数値計算法」, サイエンス社, 2001.
[9] 峯村吉泰,「C と Java で学ぶ数値シミュレーション入門」, 森北出版, 2003.
[10] 市田浩三, 吉本富士市,「スプライン関数とその応用」, 教育出版, 1982.
[11] 菅野敬祐, 吉村和美, 高山文雄,「C によるスプライン関数」, 東京電機大学出版局, 2000.
[12] 南茂夫,「科学計測のための波形データ処理」, CQ 出版株式会社, 1996.
[13] 岩谷宏 訳,「速習 JavaSwing プログラミング」, ソフトバンクパブリッシング, 1999.
[14] 林晴比古,「新 Java 言語入門シニア編」, ソフトバンクパブリッシング, 2003.
[15] 芹沢浩,「Java グラフィックス完全制覇」, 技術評論社, 2002.
[16] 柏原正三,「JavaGUI コンポーネント完全制覇」, 技術評論社, 2002.
[17] 奥村晴彦ほか,「Java によるアルゴリズム事典」, 技術評論社, 2003.
[18] 峯村吉泰,「Java によるコンピュータグラフィックス」, 森北出版, 2003.
[19] 西家正起,「コンピュータグラフィックス中核系 (GKS) 機能記述」, 日本規格協会, 1990.
[20] 村田健郎, 名取亮, 唐木幸比古,「大型数値シミュレーション」, 岩波書店, 1990.
[21] 小国力 編,「行列計算ソフトウェア」, 丸善, 1991.
[22] 森口繁一, 宇田川銈久, 一松信,「数学公式 I ── 微分積分・平面曲線 ──」, 岩波書店, 1963.
[23] 森口繁一, 宇田川銈久, 一松信,「数学公式 II ── 級数・フーリエ解析 ──」, 岩波書店, 1963.
[24] 森口繁一, 宇田川銈久, 一松信,「数学公式 III ── 特殊関数 ──」, 岩波書店, 1963.

索引

■ 英数字

1 階常微分方程式　95
2 次
　──曲線　144
　──元開曲線　209
　──元閉曲線　214
　──方程式　58
2 分法　63
3 次
　──元極座標　176
　──スプライン補間　52
　──の B スプライン　43
　──方程式　61
8 進数　224
16 進数　224

Applet　229

break　228
B スプライン　138, 207

continue　228

DFT　172

else　228
else if　228
extends　226

FFT　172
for　228
Fortran　217

html 文書　228

if　228
import　229, 233
init　229

JApplet　229
javadoc　228

link　218
LU 分解　116

main　229
Math　228

new　226

package　233
public　225

QR
　──分解　161
　──法　160

return　228

Smalltalk　217
static　225
super　226
swing　229
switch　228

this　235
throws　220

while　228

X 線　181

■ あ

アダムス・ムルトン法　101
アプリケーション　229
アプレット　229
　──ビューア　231
安定なアルゴリズム　10

一段階法　97
インクリメント　227
インタフェース　74, 234

上三角行列　117
打ち切り誤差　1

エルミート
　──行列　147
　──の補間法　39

オイラー
　──の公式　169
　──の前進公式　97
オーバーライド　226
オーバーロード　226
オブジェクト　217
　──指向　217

——ファイル　218

■ か

ガウス
——型積分　81
——関数　180
——関数のフーリエ変換　171
——の消去法　112
——・エルミートの公式　90
——・ラゲールの公式　88
——・ルジャンドルの公式　82
拡張 B スプライン　50
仮想浅間降灰予測　202
カルダノ公式　61
関係演算子　227
関数近似　125
完全
——系　177
——直交規格関数系　177

キーワード　223
規格化された B スプライン　44
刻み幅誤差　2
軌道計算　103
ギブンスの直交行列　160
基本データ型　224
逆 2 次関数法　70
逆関数法　190
逆行列　123
逆フーリエ変換　170
キャスト演算子　220, 227
吸収曲線　181
級数展開　2, 168
球ベッセル関数　5, 178
球面
——一様分布乱数　195
——調和関数　177
行列式　119
——の値　120
金属銅　181

クイックソート　20
矩形
——近似　75
——波　166
櫛ソート　15
区分的多項式　43
雲形定規　205
クラマーの公式　119
クロネッカーのデルタ　147, 177

継承　226
係数行列　112
桁落ち誤差　4

高階常微分方程式　101
構造体　61, 219
高速フーリエ変換　172
後退代入　112
固有
——値　147
——値問題　144, 147
——ベクトル　147
——方程式　148
コンストラクタ　225, 226
コンボリューション　180, 181

■ さ

最確値　125
最小二乗法　125
最大値　12
作図　205
サビツキー・ゴーレイ法　135
三角関数　168
三項条件演算子　227
算術平均　126
参照型　224

シェルソート　18
自在定規　205
指数関数　169
下三角行列　117
実装　234
シミュレーション　200
シャッフル　10
周期
——関数　54
——スプライン　56
主軸問題　144
シュレーディンガー方程式　106
仕様書　228
常微分方程式　95
人工衛星　103
シンプソンの公式　2, 77, 101

水素原子　106
数値積分法　75
スプライン補間　43

正規分布　125
——乱数　193
制御文　228
整数　224
正則行列　120
積分　188
切断べき関数　205
漸化式　5
線形関係　126
前進消去　112
選択ソート　16

挿入ソート　17
ソースプログラム　218
疎行列　157
属性　226

■ た

第 1 添字　224, 225
第 2 添字　224, 225

対角行列　148
台形
　──近似　76
　──公式　2, 101
対称行列　147
代数方程式　58
代入演算子　227
楕円　144
多重積分　190
畳み込み　180, 181
　──除去　180, 185
多段階法　100
単位行列　123
単精度　224

チェビシェフ
　──多項式　130
　──の積分公式　86
重複
　──データ点　213
　──節点　207
直接法　148
直交
　──規格関数系　177
　──行列　147
　──変換　147

ディクリメント　227
ディコンボリューション　180
テーラー展開　97
手続き型　217
デルタ関数　177
転置行列　147
天頂角　176

動径　176
　──運動量波動関数　179
　──関数　177
　──波動関数　179
　──フーリエ変換　179
統計誤差　180
特殊文字　223
ドュブア・コックス　43

■ な

内部クラス　225

入出力　219
ニュートン
　──の補間法　37
　──・コーツ型積分　76
　──・ラフソン法　67

鋸波　166

■ は

バーチャルマシン　219
媒介変数　51, 209
倍精度　224
配列　224
　──要素　224
ハウスホールダー法　157
掃き出し法　115
挟み撃ち法　64
パッケージ　233
波動関数　106
バブルソート　13
半値幅　181
反復法　67, 70

ヒープ　22
　──ソート　22
非周期スプライン　56
ヒット・オア・ミス法　190
微分値　34, 39
ピボット　113
ビュフォンの針　200
標準
　──指数分布乱数　190
　──偏差　171
表読み取り法　196

フィールド　225
フーリエ
　──級数　166
　──変換　166, 170
ブール　224
不確定性関係　180
複素数型　224
節点　43, 205
浮動小数点数　224
部分波展開　178

平滑化　125, 135, 180
平均値の定理　95
平行六面体　121
平面曲線図形　51
偏角　176

ホイットニの条件　47
方位角　176
方程式の解法　58
補間法　29
ボックス・マーラー法　193

■ ま

マージ　24
　──ソート　26
丸め誤差　7

密行列　157
ミルン法　101

メソッド　225

文字　223
　──型　224
　──列　224
モンテカルロ法　188

■ や

ヤコビ法　149

ユニタリー行列　148

要素数　225
予測子・修正子法　100
予約語　223

■ ら

ライブラリ　218, 233
ラグランジュの補間法　29
乱数　188
ランチョス法　157

リーゼンフェルトの方法　208
離散フーリエ変換　172
リテラル　223
リンカ　218
リンク　218

累乗法　153
累積分布関数　190
ルジャンドル陪関数　177
ルンゲ・クッタ
　——法 2 次公式　98
　——法 4 次公式　100
　——・ジル法　105

例外処理　220
レギュラ・ファルシ法　64
連結演算子　227
連立
　—— 1 階常微分方程式　102
　—— 1 次方程式　111
　——同次方程式　148

ロードモジュール　218
ローレンツ
　——型分布乱数　192
　——関数　180
論理
　——演算子　227
　——型　224

＜著者紹介＞

和光システム研究所

和光信也（わこう しんや）

学　歴　東京大学理学部物理学科卒業（1961年）
　　　　理学博士（東京大学数物系大学院）（1966年）
職　歴　東京大学助手，東京大学助教授，図書館情報大学教授，筑波大学教授
現　在　筑波大学名誉教授
　　　　埼玉工業大学非常勤講師
　　　　(株）和光システム研究所
　　　　http://www11.plala.or.jp/wakoh/index.htm
著　書　『コンピュータでみる 固体の中の電子』（講談社，1992年）

松本　勲（まつもと いさお）

学　歴　埼玉工業大学工学部電子工学科卒業（1994年）
　　　　図書館情報大学修士課程修了（1997年）
　　　　博士（理学，総合研究大学院大学）(2000年）
職　歴　高エネルギー加速器研究機構 研究機関研究員
現　在　(株）和光システム研究所

Javaで学ぶ数値解析

2005年　4月10日　第1版1刷発行

著　者　和光システム研究所

学校法人　東京電機大学
発行所　東京電機大学出版局
代表者　加藤康太郎

〒101-8457
東京都千代田区神田錦町2-2
振替口座 00160-5- 71715
電話（03) 5280-3433（営業）
　　（03) 5280-3422（編集）

制作　(株) グラベルロード
印刷　新灯印刷 (株)
製本　渡辺製本 (株)
装丁　高橋壮一

ISBN4-501-53920-8　C3004